Emac API Specification

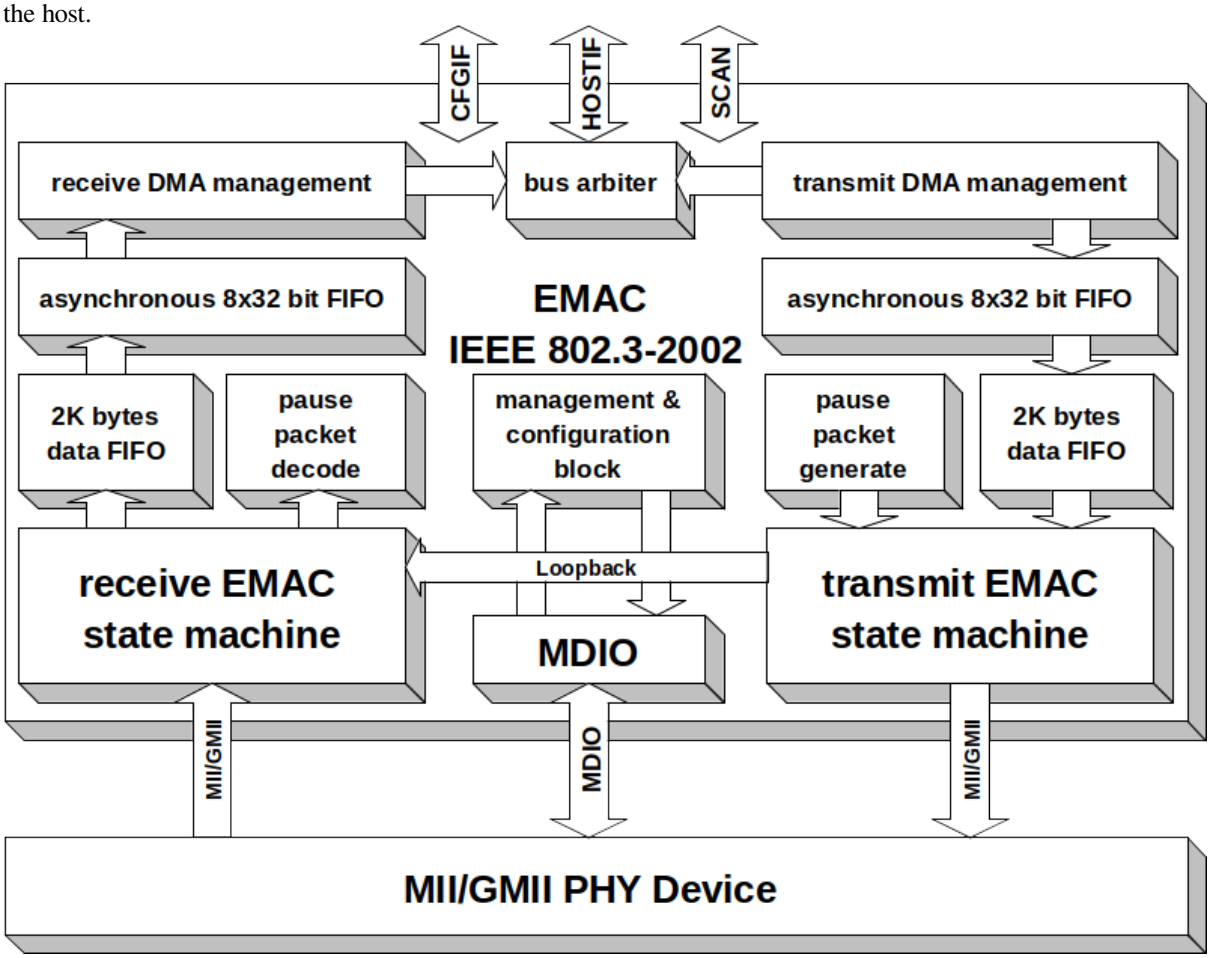
Jan 19, 2024

OVERVIEW DOCUMENTATION

CHAPTER
ONE

DESCRIPTION

EMAC is a configurable, fully compatible IEEE 802.3–2002 implementation of Media Access Controller interface operating at 10/100/1000 Mbps with integrated data FIFO's, configuration registers and Tulip DMA host interface. It provides standard MII/GMII and MDIO interfaces to all compliant PHY devices and a generic 32-bit interface to the host.



CHAPTER
TWO

FEATURES

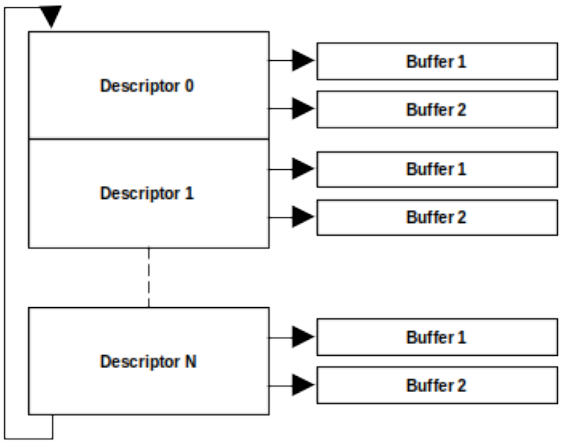
- Compatible with the IEEE 802.3–2002 standard
- Configurable 10/100/1000 Mbps speed
- IEEE Std 802.3-2002 compliant Media Independent Interface for connection to external 10/100 Mbps PHY transceivers
- IEEE Std 802.3-2002 compliant Gigabit Media Independent Interface for connection to external 1000 Mbps PHY transceivers
- Supports 10BASE-T and 100BASE-TX/FX IEEE Std 802.3-2002 compliant MII PHY's at full or half duplex operating modes
- Supports Gigabit Ethernet and 1000BASE-T IEEE Std 802.3-2002 compliant GMII PHY's at full or half duplex operating modes
- Supports MDIO management control writes and reads with the PHY's
- Configurable Full/Half Duplex for any speed
- Supports burst operation and carrier extend when 1000 Mbps operating mode is selected
- CSMA/CD compliant operation at 10 Mbps, 100 Mbps, 1000 Mbps in half duplex mode
- Pause frame capability IEEE 802.3x compliant
- Backpressure half duplex flow control algorithm
- Supports Jumbo frame transfer, up to 16KB, both receive and transmit
- Configurable address filtering modes: 16 perfect addresses, 512 hash-filtered multicast addresses and one perfect address, inverse perfect filtering
- Supports unicast, multicast, and broadcast
- Supports promiscuous address receive mode
- Provides auto pad and Frame Check Sequence field insertion for transmit operation
- Provides auto Frame Check Sequence field checking and removal for receive operation
- Programmable interframe gap
- Internal FIFO's for TX/RX data flows (configurable size); implemented in Dual port RAM
- Internal loop-back capability
- 32-bit Host interface supporting DMA descriptor base system (Tulip Driver) with one general interrupt line
- Configurable DMA transfer burst length
- Big/little endian for DMA data transfers
- Descriptor/buffer architecture, supporting ring/chain data structures
- Automatic descriptor polling
- Contains Control and Status Registers Block (CSR)
- Provides global and per frame statistics for both transmit and receive
- Supports a wide range of Host clock frequencies
- Low power capability for all internal blocks independently (gating clock mechanism used when no activity)

CHAPTER
THREE

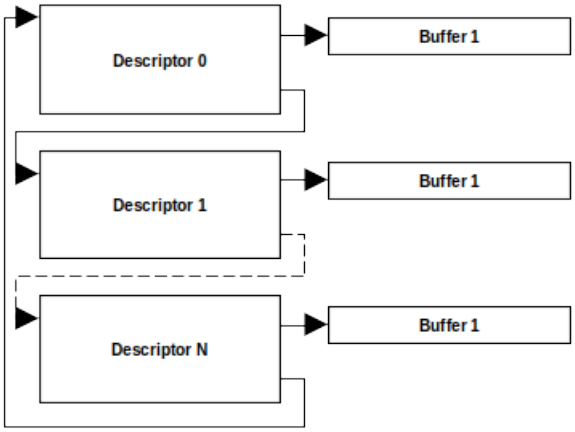
PROGRAMMING

3.1 Descriptor Lists and Data Buffers

The EMAC transfers frame data to and from receive and transmit buffers in host memory. The descriptors resides also in the host memory and acts as pointers to these buffers.
There are two descriptor lists, one for the receive buffers and one for transmit. The base address of each list is written into CSR3 and CSR4, respectively. A descriptor list is forward-linked, either implicitly or explicitly. The last descriptor may point back to the first entry to create a ring structure. Explicit chaining of descriptors is accomplished by setting the second address chained in both receive and transmit descriptors. The descriptor lists reside in the host physical memory address space. Each descriptor can point to a maximum of two buffers. This enables two buffers to be used, physically addressed, and not contiguous in memory.
A data buffer consists of either an entire frame or part of a frame, but it cannot exceed a single frame. Buffers contain only data. The buffer status is maintained in the descriptor. Data chaining refers to frames that span multiple data buffers. Data chaining can be enabled or disabled. The data buffers also reside in the host physical memory space.
Descriptor Ring Configuration:



Descriptor Chain Configuration:



3.2 Transmit descriptors

Descriptors and receive buffer addresses must be 32-bit word aligned.
Providing two buffers, two byte-count buffers, and two address pointers in each descriptor enables the adapter port to be compatible with various types of memory management schemes.
Transmit Descriptor Format:

	31			0
TDES0	OWN	Status		
TDES1	Control	Buffer 2 Size	Buffer 1 Size	
TDES2	Buffer Address 1			
TDES3	Buffer Address 2			

3.3 Receive descriptors

Descriptors and receive buffer addresses must be 32-bit word aligned.
Providing two buffers, two byte-count buffers, and two address pointers in each descriptor enables the adapter port to be compatible with various types of memory management schemes.
Receive Descriptor Format:

	31			0
RDES0	OWN	Status		
RDES1	Control	Buffer 2 Size	Buffer 1 Size	
RDES2	Buffer Address 1			
RDES3	Buffer Address 2			

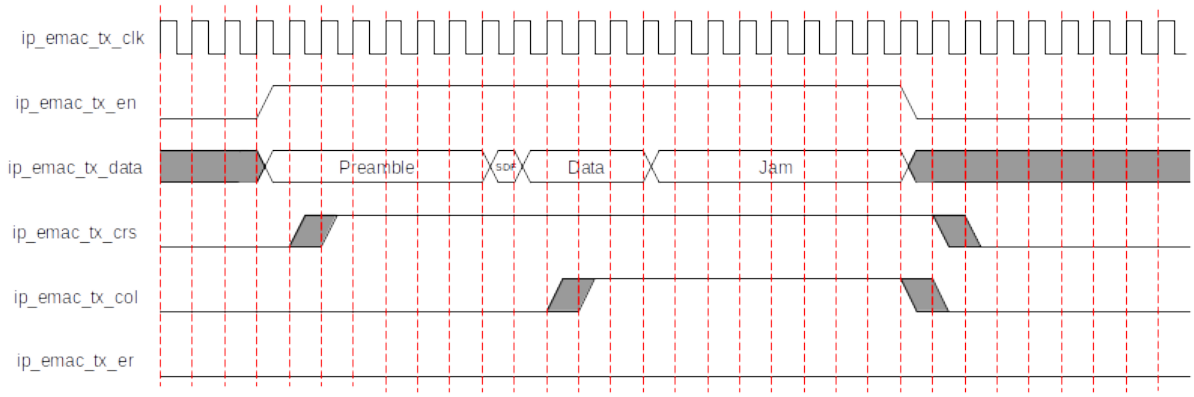
CHAPTER
FOUR

TIMING

All waveforms given in this section illustrate the behavior of the EMAC block assuming a typical IEEE 802.3-2002 behavior for the remote port.

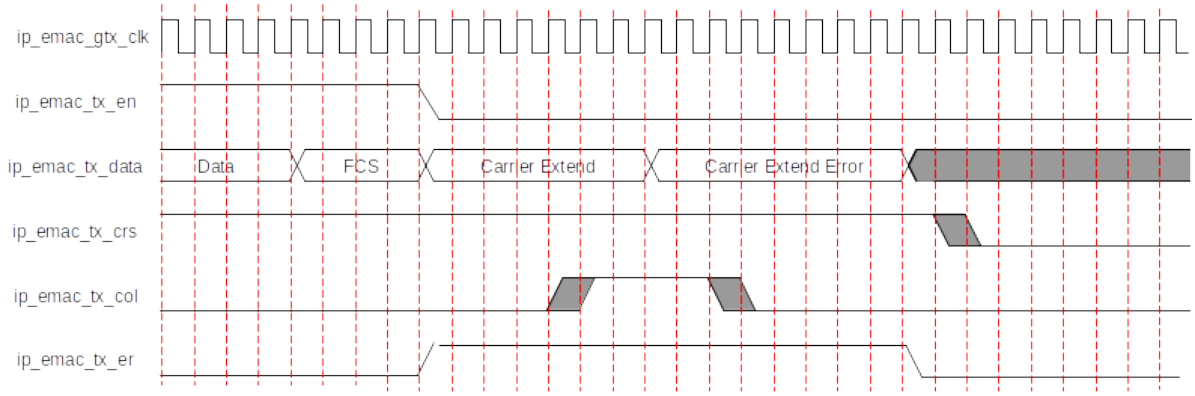
4.1 Early Collision Timing

10/100Mbps Early Collision Behavior:



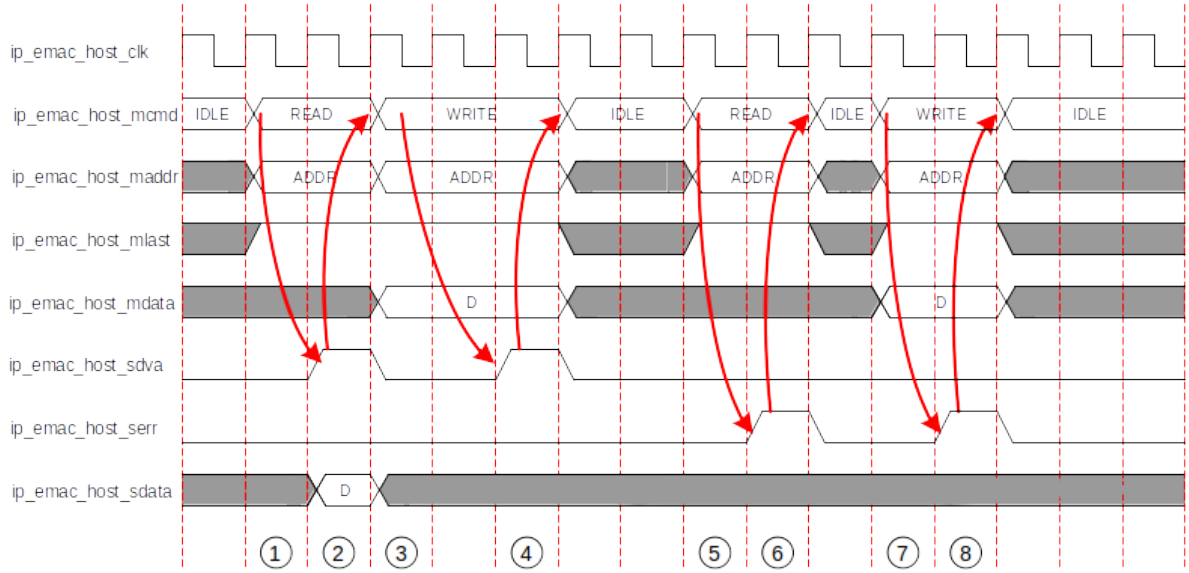
This figure describes a MII early collision behavior. The packet is retransmitted after defer and backoff period.

1000Mbps Early Collision Behavior:



This figure describes a GMII early collision behavior. The packet is retransmitted after defer and backoff period.

4.2 Host Master Interface Single Read/Write Timing



This figure presents the host interface timing when single commands are used. The *ip_emac_host_mlast* asserted high indicates that single command is issued. The *ip_emac_host_mcmd* transition to IDLE is determined by the assertion of *ip_emac_host_sdva* or *ip_emac_host_serr* signal. After the slave response for the current command the next command is loaded by the *ip_emac_host_mcmd* if available, else the IDLE state is loaded.

1. The EMAC issues a single READ command
2. The slave responds by asserting *ip_emac_host_sdva* and *ip_emac_host_sdata*
3. The EMAC issues a single WRITE command
4. The slave accepts the WRITE command by asserting *ip_emac_host_sdva*
5. The EMAC issues a single READ command
6. The slave responds by asserting *ip_emac_host_serr* (the transfer cannot be complete by the slave)
7. The EMAC issues a single WRITE command
8. The slave responds by asserting *ip_emac_host_serr* (the transfer cannot be complete by the slave)

CHAPTER
FIVE

DESIGN TOPS

5.1 Module ip_emac_top

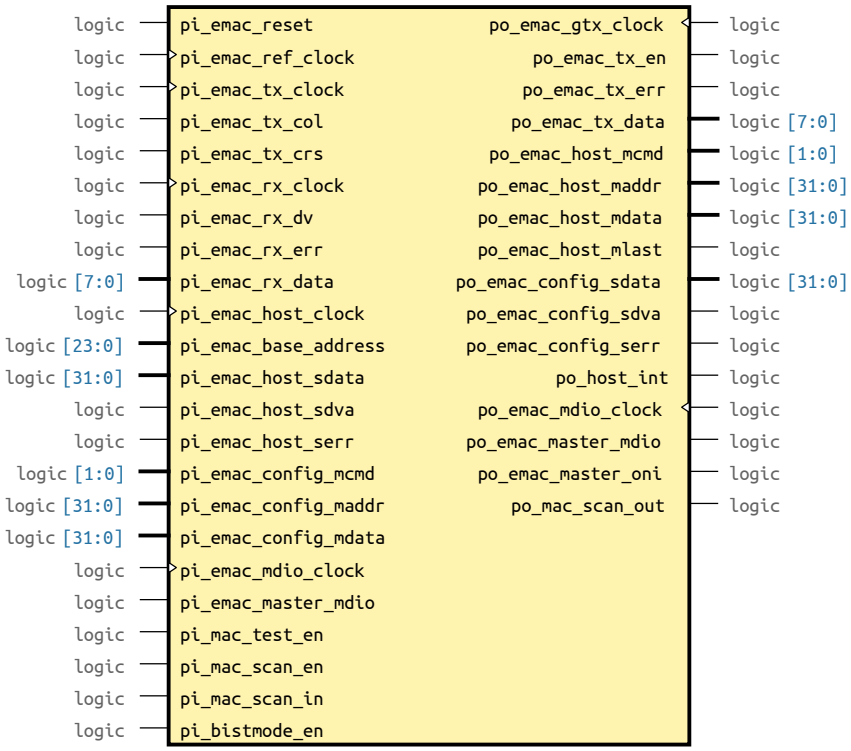


Fig. 1: Block Diagram of ip_emac_top

Table 1: Ports

Name	Direction	Type	Description
pi_emac_reset	input	wire logic	Global Hardware reset (active low)
pi_emac_ref_clock	input	wire logic	GMII 125 MHz reference clock
pi_emac_tx_clock	input	wire logic	Transmit GMII/MII interface Transmit MII 25/2.5 MHz clock (from PHY)
po_emac_gtx_-clock	output	wire logic	Transmit GMII 125 MHz clock (to PHY)
po_emac_tx_en	output	wire logic	Transmit MII/GMII enable indication (to PHY)
po_emac_tx_err	output	wire logic	Transmit MII/GMII error indication (to PHY)
po_emac_tx_data	output	wire logic[7:0]	Transmit MII/GMII data (MII data is po_-emac_tx_data[3:0]) (to PHY)
pi_emac_tx_col	input	wire logic	Collision indication (from PHY)

continues on next page

Table 1 – continued from previous page

Name	Direction	Type	Description
pi_emac_tx_crs	input	wire logic	Carrier Sense indication (from PHY)
pi_emac_rx_clock	input	wire logic	Receive GMII/MII interface Receive GMII/MII 125/25/2.5 MHz clock (from PHY)
pi_emac_rx_dv	input	wire logic	Receive MII/GMII data valid indication (from PHY)
pi_emac_rx_err	input	wire logic	Receive MII/GMII error indication (from PHY)
pi_emac_rx_data	input	wire logic[7:0]	Receive MII/GMII data (MII data is pi_emac_rx_data[3:0]) (from PHY)
pi_emac_host_clock	input	wire logic	HOST interface (common)
pi_emac_base_address	input	wire logic[23:0]	host data interface
po_emac_host_mcmd	output	wire logic[1:0]	
po_emac_host_maddr	output	wire logic[31:0]	
po_emac_host_mdata	output	wire logic[31:0]	
po_emac_host_mlast	output	wire logic	
pi_emac_host_sdata	input	wire logic[31:0]	
pi_emac_host_sdva	input	wire logic	
pi_emac_host_serr	input	wire logic	
pi_emac_config_mcmd	input	wire logic[1:0]	host config interface
pi_emac_config_maddr	input	wire logic[31:0]	
pi_emac_config_mdata	input	wire logic[31:0]	
po_emac_config_sdata	output	wire logic[31:0]	pi_emac_config_mlast ,
po_emac_config_sdva	output	wire logic	
po_emac_config_serr	output	wire logic	
po_host_int	output	wire logic	general interrupt to host
pi_emac_mdio_clock	input	wire logic	MDIO interface
po_emac_mdio_clock	output	wire logic	
pi_emac_master_mdio	input	wire logic	
po_emac_master_mdio	output	wire logic	
po_emac_master_oni	output	wire logic	
pi_mac_test_en	input	wire logic	Test and Scan interface signals
pi_mac_scan_en	input	wire logic	
pi_mac_scan_in	input	wire logic	
po_mac_scan_out	output	wire logic	
pi_bistmode_en	input	wire logic	

Submodules

- **ip_emac_top**
  - host_clk_mng : *ip_host_clk_mng_g*
  - host_if : *ip_mac_hostif_top*
  - mac_top : *ip_mac_top_g* #(.TX_MEM_ADDR(10), .RX_MEM_ADDR(10))
  - regs_bank : *ip_mac_regs_bank*

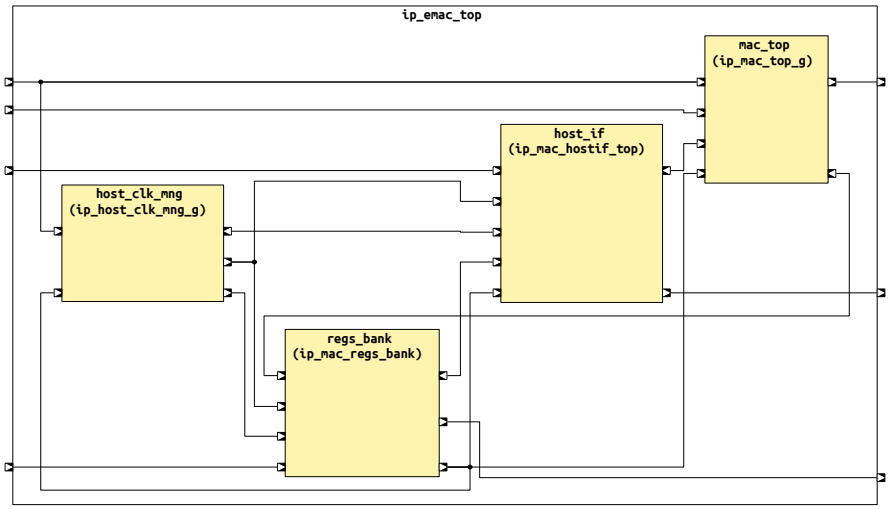


Fig. 2: Flow Diagram of ip_emac_top

CHAPTER
SIX

MODULES

6.1 Module cts_buffer

Fig. 1: Block Diagram of cts_buffer

Table 1: Ports

Name	Direction	Type	Description
cts_buff_in	input	wire logic	a CTS buffer input clock
cts_buff_out	output	wire logic	a CTS buffer output clock

Instances

- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `output_cts`

6.2 Module dff_metastable

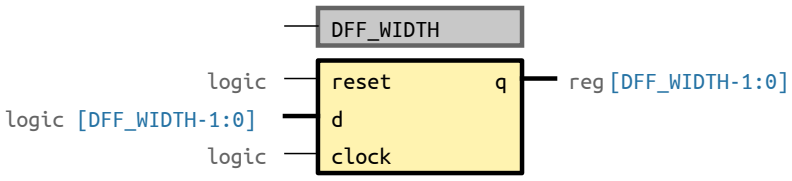


Fig. 2: Block Diagram of dff_metastable

Table 2: Parameters

Name	Default value	Description
DFF_WIDTH	1	

Table 3: Ports

Name	Direction	Type	Description
reset	input	wire logic	
d	input	wire logic[DFF_WIDTH-1:0]	
clock	input	wire logic	
q	output	var reg[DFF_WIDTH-1:0]	

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_rd : dff_metastable#(.DFF_WIDTH(1))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > `metastable_toggle_wr : dff_metastable#(.DFF_WIDTH(1))`

Submodules

- **dff_metastable#(.DFF_WIDTH(1))**
  - dffrhqx1_0 : *dffrhqx1*

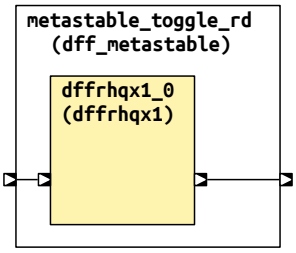


Fig. 3: Flow Diagram of dff_metastable

6.3 Module dffrhqx1

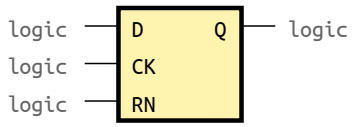


Fig. 4: Block Diagram of dffrhqx1

Table 4: Parameters

Name	Default value	Description
tplhRNQ	1.0	
tphlRNQ	1.0	
tplhCKQ	1.0	
tphlCKQ	1.0	
tsetup$D$CK	1.0	
thold$D$CK	1.0	
trec$RN	1.0	
tminpwl$RN	1.0	
tminpwl$CK	1.0	
tminpwh$CK	1.0	

Table 5: Ports

Name	Direction	Type	Description
Q	output	wire logic	
D	input	wire logic	
CK	input	wire logic	
RN	input	wire logic	

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > *ip_sync_cell* > *dff_metastable* > `dffrhqx1_0`

6.4 Module dram_001

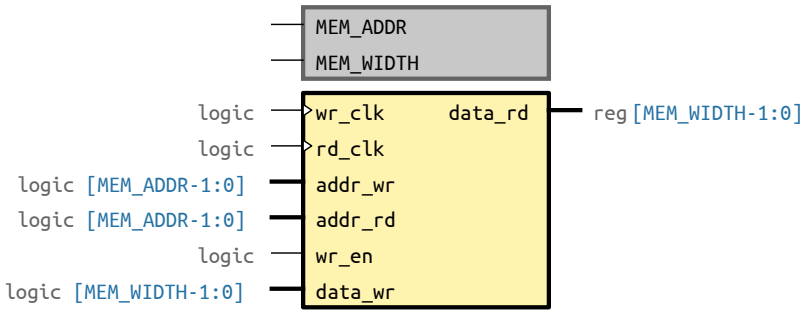


Fig. 5: Block Diagram of dram_001

Table 6: Parameters

Name	Default value	Description
MEM_ADDR	5	Address width (9 -> 512 locations, 10->1024)
MEM_WIDTH	16	Data width

Table 7: Ports

Name	Direction	Type	Description
wr_clk	input	wire logic	Write clock
rd_clk	input	wire logic	Read clock
addr_wr	input	wire logic[MEM_-ADDR-1:0]	Write address
addr_rd	input	wire logic[MEM_-ADDR-1:0]	Read address
wr_en	input	wire logic	Write enable
data_wr	input	wire logic[MEM_-WIDTH-1:0]	Write data
data_rd	output	var reg[MEM_-WIDTH-1:0]	Read data (registered)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_mac_dram_002* > `dram_001 : dram_001#(.MEM_ADDR(10), .MEM_WIDTH(37))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_mac_dram_004* > `dram_001 : dram_001#(.MEM_ADDR(4), .MEM_WIDTH(32))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_mac_dram_003* > `dram_001 : dram_001#(.MEM_ADDR(4), .MEM_WIDTH(16))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_mac_dram_001* > `dram_001 : dram_001#(.MEM_ADDR(10), .MEM_WIDTH(39))`

6.5 Module gate_clock_cell_g

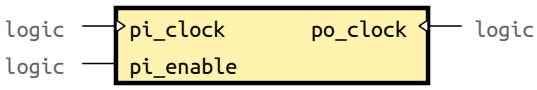


Fig. 6: Block Diagram of gate_clock_cell_g

Table 8: Ports

Name	Direction	Type	Description
pi_clock	input	wire logic	Input functional clock
po_clock	output	wire logic	Output gated clock (multiplexed with test clock)
pi_enable	input	wire logic	Enable output clock

Instances

- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > *ip_gate_clock_g* > `gate_clock`

6.6 Module ip_async_fifo_g

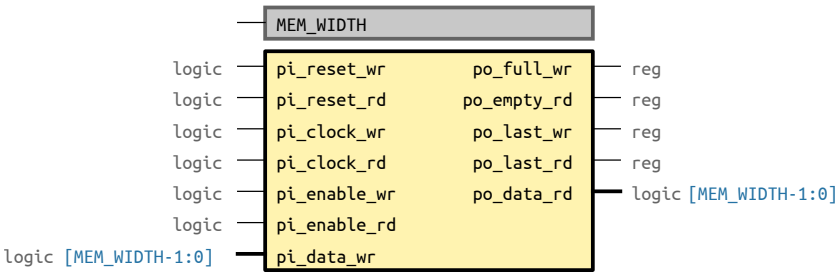


Fig. 7: Block Diagram of ip_async_fifo_g

Overview

Asynchronous FIFOs are used in designs to safely pass multi-bit data words from one clock domain to another.

Data words are placed into a FIFO buffer memory array by: - control signals in one clock domain - and the data words are removed from another port of the same FIFO buffer memory array by control signals from a second clock domain.

Table 9: Parameters

Name	Default value	Description
MEM_WIDTH	32	Data width

Table 10: Ports

Name	Direction	Type	Description
pi_reset_wr	input	wire logic	Write synchronous reset
pi_reset_rd	input	wire logic	Read synchronous reset
pi_clock_wr	input	wire logic	Write clock
pi_clock_rd	input	wire logic	Read clock
pi_enable_wr	input	wire logic	Write enable
pi_enable_rd	input	wire logic	Read enable
po_full_wr	output	var reg	FIFO full indication
po_empty_rd	output	var reg	FIFO empty indication
po_last_wr	output	var reg	FIFO last location for write (almoast full)
po_last_rd	output	var reg	FIFO last location for read (almoast empty)
pi_data_wr	input	wire logic[MEM_-WIDTH-1:0]	FIFO data input
po_data_rd	output	wire logic[MEM_-WIDTH-1:0]	FIFO data output

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_async : ip_async_fifo_g#(.MEM_WIDTH(37))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_data_async : ip_async_fifo_g#(.MEM_WIDTH(39))`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_stat_async : ip_async_fifo_g#(.MEM_WIDTH(10))`

Submodules

- **ip_async_fifo_g#(.MEM_WIDTH(37))**
  - cell_0 : *ip_sync_cell*
  - cell_1 : *ip_sync_cell*
  - cell_2 : *ip_sync_cell*
  - cell_3 : *ip_sync_cell*
  - cell_4 : *ip_sync_cell*
  - cell_5 : *ip_sync_cell*
  - cell_6 : *ip_sync_cell*
  - cell_7 : *ip_sync_cell*
- **ip_async_fifo_g#(.MEM_WIDTH(39))**
  - cell_0 : *ip_sync_cell*
  - cell_1 : *ip_sync_cell*
  - cell_2 : *ip_sync_cell*
  - cell_3 : *ip_sync_cell*
  - cell_4 : *ip_sync_cell*
  - cell_5 : *ip_sync_cell*
  - cell_6 : *ip_sync_cell*
  - cell_7 : *ip_sync_cell*
- **ip_async_fifo_g#(.MEM_WIDTH(10))**
  - cell_0 : *ip_sync_cell*
  - cell_1 : *ip_sync_cell*
  - cell_2 : *ip_sync_cell*
  - cell_3 : *ip_sync_cell*
  - cell_4 : *ip_sync_cell*
  - cell_5 : *ip_sync_cell*
  - cell_6 : *ip_sync_cell*
  - cell_7 : *ip_sync_cell*

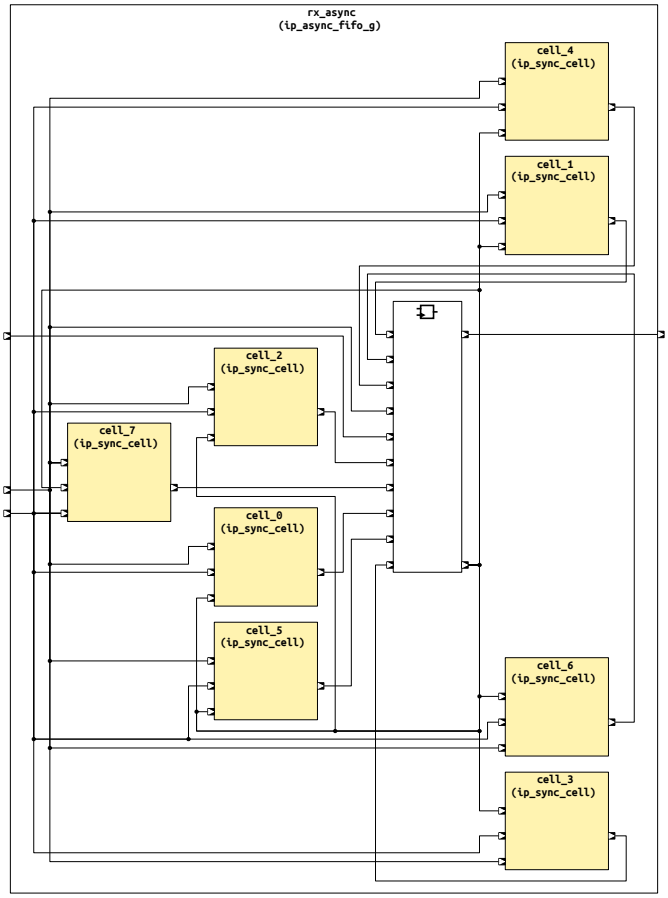


Fig. 8: Flow Diagram of ip_async_fifo_g

6.7 Module ip_gate_clock_g

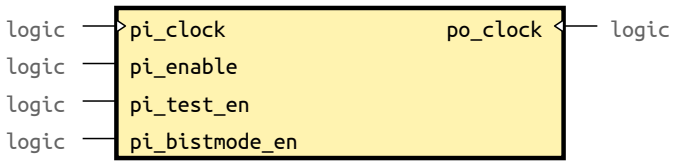


Fig. 9: Block Diagram of ip_gate_clock_g

Table 11: Ports

Name	Direction	Type	Description
pi_clock	input	wire logic	Input free running clock
pi_enable	input	wire logic	Enable output clock
pi_test_en	input	wire logic	Test enable
pi_bistmode_en	input	wire logic	
po_clock	output	wire logic	Output free running clock

Instances

- *ip_emac_top* > *ip_host_clk_mng_g* > `host_free_clock`
- *ip_emac_top* > *ip_host_clk_mng_g* > `host_gate_clock_1`
- *ip_emac_top* > *ip_host_clk_mng_g* > `host_gate_clock_2`
- *ip_emac_top* > *ip_host_clk_mng_g* > `host_gate_clock_3`
- *ip_emac_top* > *ip_host_clk_mng_g* > `host_gate_clock_4`
- *ip_emac_top* > *ip_host_clk_mng_g* > `host_gate_clock_5`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `gtx_clock_out`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `host_free_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `host_gate_clock_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `mdio_free_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `rx_free_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `rx_gate_clock_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `rx_gate_clock_2`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `rx_gate_clock_3`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_free_clock`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_gate_clock_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_gate_clock_2`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_gate_clock_3`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_gate_clock_4`

Submodules

- **ip_gate_clock_g**
  - gate_clock : *gate_clock_cell_g*
  - output_cts : *cts_buffer*

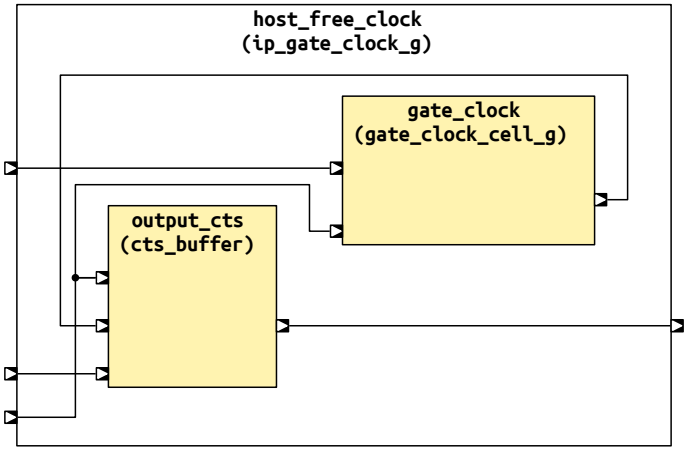


Fig. 10: Flow Diagram of ip_gate_clock_g

6.8 Module ip_host_clk_mng_g

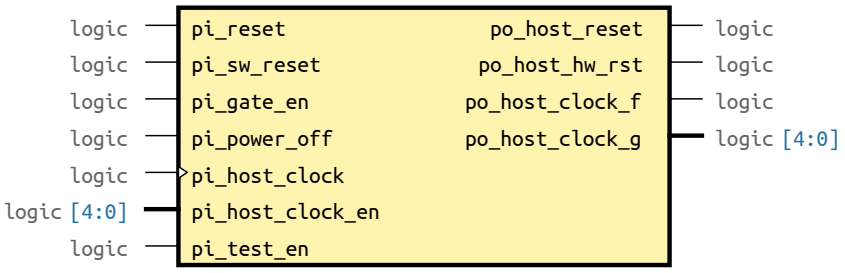


Fig. 11: Block Diagram of ip_host_clk_mng_g

Table 12: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Resets Hardware reset (active low)
pi_sw_reset	input	wire logic	Software reset (active high)
pi_gate_en	input	wire logic	Auto Gating Clock Enable (power saving)
po_host_reset	output	wire logic	Hardware/Software Global output reset (HOST clock domain)
po_host_hw_rst	output	wire logic	Hardware Global output reset (HOST clock domain)
pi_power_off	input	wire logic	Power OFF Power off (all internal clocks are disabled)
pi_host_clock	input	wire logic	Clocks HOST clock used by the reset synchronization block
pi_host_clock_en	input	wire logic[4:0]	HOST clock enable
po_host_clock_f	output	wire logic	Output Host Clock Host Free Clock
po_host_clock_g	output	wire logic[4:0]	Host Gated Clock
pi_test_en	input	wire logic	Test and Scan interface signals Test enable (test clock select)

Instances

- *ip_emac_top* > `host_clk_mng`

Submodules

- **ip_host_clk_mng_g**
  - host_free_clock : *ip_gate_clock_g*
  - host_gate_clock_1 : *ip_gate_clock_g*
  - host_gate_clock_2 : *ip_gate_clock_g*
  - host_gate_clock_3 : *ip_gate_clock_g*
  - host_gate_clock_4 : *ip_gate_clock_g*
  - host_gate_clock_5 : *ip_gate_clock_g*
  - host_sreset : *ip_sync_reset_g*
  - hw_host_sreset : *ip_sync_reset_g*

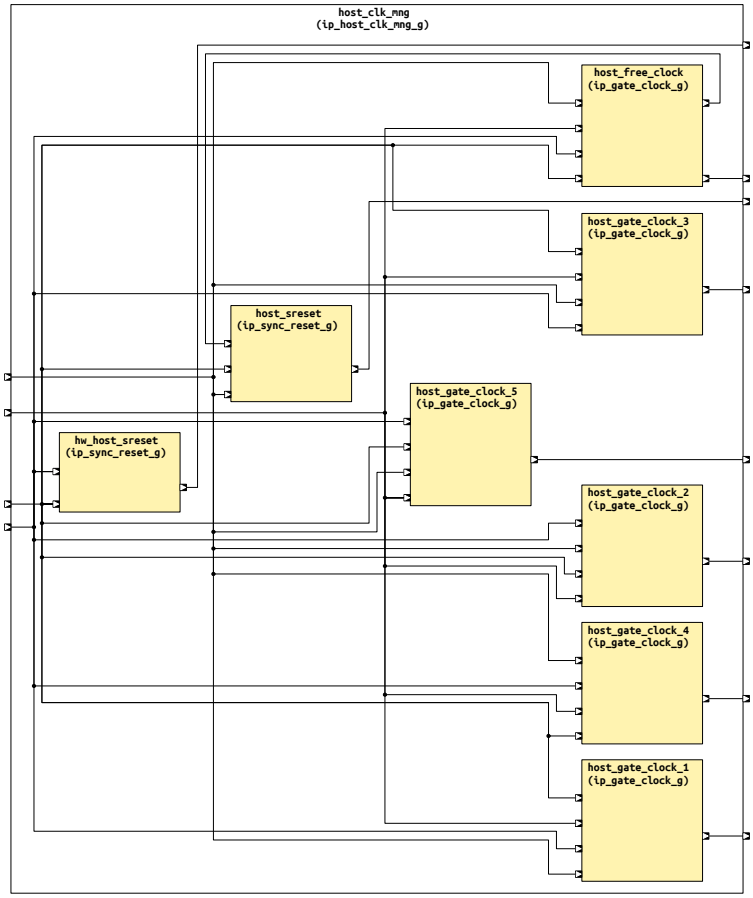


Fig. 12: Flow Diagram of ip_host_clk_mng_g

6.9 Module ip_mac_big_endian

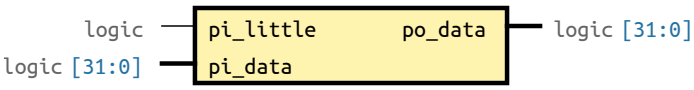


Fig. 13: Block Diagram of ip_mac_big_endian

Overview

Data should be translated to little *endian format*. The following module is a combinatorial module and is responsible to translate **a 32-bit wide data bus** (4-byte word) big endian format into a little endian organized word. When the input data is *little endian* organized the data is passed trough this block and remain unchanged.

Table 13: Ports

Name	Direction	Type	Description
pi_little	input	wire logic	Configuration Little endian
pi_data	input	wire logic[31:0]	Input little/big endian 32-bit word Input 32-bit data
po_data	output	wire logic[31:0]	Output little endian 32-bit word Output 32-bit translated data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_endian`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_endian`

6.10 Module ip_mac_cfg_hash_g

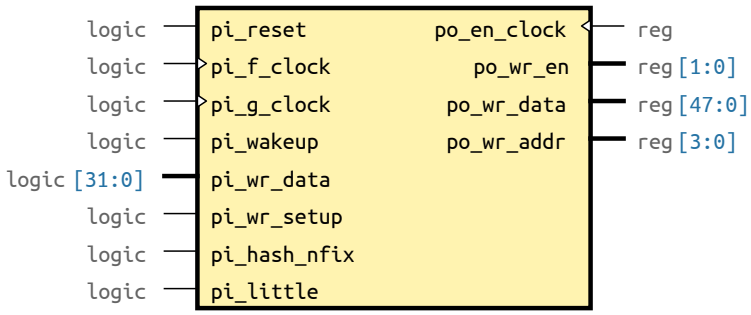


Fig. 14: Block Diagram of ip_mac_cfg_hash_g

Overview

A special configuration sequence must be performed before the reception process is started, except when it operates in promiscuous filtering mode. This is done by successively writing to CRFT configuration register. Depending on the selected filtering mode (CSR6[5:4] Filtering status field), the appropriate information should be written into the CRFT (see ***EMAC Functional Specification 1.1***)

When hash multicast or hash filtering operating mode is selected, 16 consecutive writes should be performed to the CRFT register, representing the 512-bit hash table information. After completing the hash table write, two additional writes should be performed on CRFT encapsulating the perfect 48-bit MAC address used for exact physical destination address match processing.

When perfect filtering mode is selected 32 consecutive writes should be performed to the CRFT register encapsulating 16 exact match addresses.

The structure of each 32 bit word written to CRFT register depends on CSR0[7] ***(Big/Little Endian field)*** and CSR6[5:4] (Filtering status field). See 9.2.1 and 9.2.2 for a detailed description of CRFT configuration sequence.

Table 14: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware Reset (host clock domain)
pi_f_clock	input	wire logic	Free HOST interface clock signal
pi_g_clock	input	wire logic	Gated HOST interface clock signal
po_en_clock	output	var reg	Enable HOST interface clock signal
pi_wakeup	input	wire logic	From/To Receive DMA Wake-up internal clock used by the Setup Frame FSM,
pi_wr_data	input	wire logic[31:0]	should be asserted at least 2 clock cycles (HOST clock) before asserting the pi_host_wr_setup (write enable) and can be de-asserted 2 clock cycles after Setup Frame complete Setup Frame data (HOST clock synchronous)
pi_wr_setup	input	wire logic	Setup Frame write enable (HOST clock synchronous)
pi_hash_nfix	input	wire logic	Configuration Hash filtering + 1 Address match / 16 Address match
pi_little	input	wire logic	Little endian
po_wr_en	output	var reg[1:0]	Hash Table Memory access Hash table write enable

continues on next page

Table 14 – continued from previous page

Name	Direction	Type	Description
po_wr_data	output	var reg[47:0]	Hash table write data
po_wr_addr	output	var reg[3:0]	Hash table write address

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Gated Clock Enable

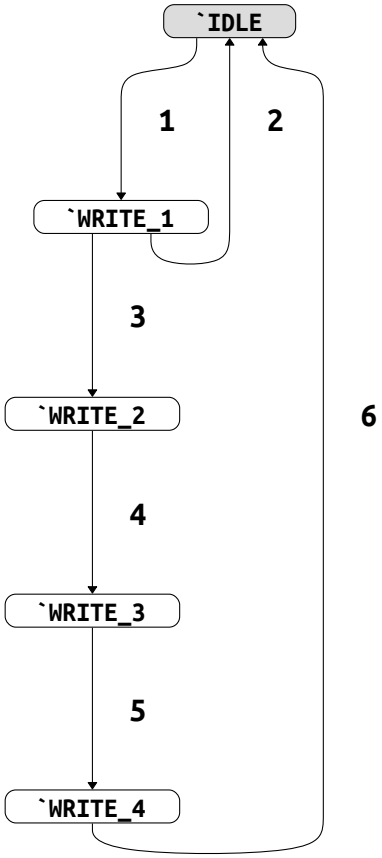


Table 15: FSM Transitions for fsm_hash_st

#	Current State	Next State	Condition
1	`IDLE	`WRITE_1	[(!(~ pi_reset) && (pi_wr_setup == 1'b1 && pi_-hash_nfix == 1'b0)), (!(~ pi_reset) && !(pi_wr_setup == 1'b1 && pi_hash_nfix == 1'b0) && (pi_wr_setup == 1'b1 && pi_hash_nfix == 1'b1))]
2	`WRITE_1	`IDLE	[(!(~ pi_reset) && (pi_wr_setup == 1'b1 && pi_-hash_nfix == 1'b0))]
3	`WRITE_1	`WRITE_2	[(!(~ pi_reset) && !(pi_wr_setup == 1'b1 && pi_-hash_nfix == 1'b0) && (pi_wr_setup == 1'b1 && pi_-hash_nfix == 1'b1) && (po_wr_addr == 4'hf))]
4	`WRITE_2	`WRITE_3	[(!(~ pi_reset) && (pi_wr_setup == 1'b1))]
5	`WRITE_3	`WRITE_4	[(!(~ pi_reset) && (pi_wr_setup == 1'b1))]
6	`WRITE_4	`IDLE	[!(~ pi_reset)]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `cfg_hash`

6.11 Module ip_mac_clk_mng_g

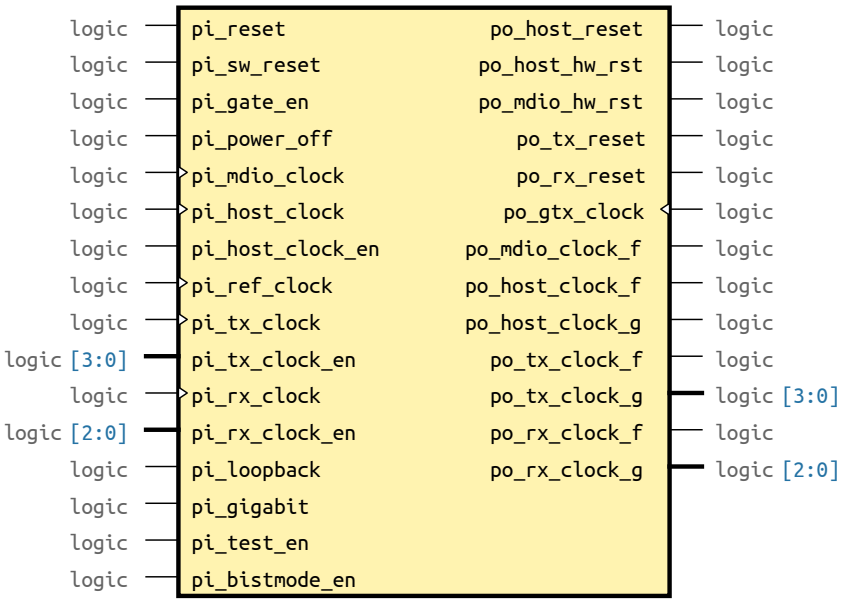


Fig. 15: Block Diagram of ip_mac_clk_mng_g

Table 16: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Resets Hardware reset (active low)
pi_sw_reset	input	wire logic	Software reset (active high)
pi_gate_en	input	wire logic	Auto Gating Clock Enable (power saving)
po_host_reset	output	wire logic	Hardware/Software Global output reset (HOST clock domain)
po_host_hw_rst	output	wire logic	Hardware Global output reset (HOST clock domain)
po_mdio_hw_rst	output	wire logic	Hardware MDIO output reset (MDIO clock domain)
po_tx_reset	output	wire logic	Hardware/Software Global output reset (transmit clock domain)
po_rx_reset	output	wire logic	Hardware/Software Global output reset (receive clock domain)
pi_power_off	input	wire logic	Power OFF Power off (all internal clocks are disabled)
pi_mdio_clock	input	wire logic	Clocks MDIO clock used by the MDIO module
pi_host_clock	input	wire logic	HOST clock used by the reset synchronization block
pi_host_clock_en	input	wire logic	HOST clock enable
pi_ref_clock	input	wire logic	125 MHz Reference Clock
po_gtx_clock	output	wire logic	GMII Transmit clock (balanced with the EMAC internal clock)
pi_tx_clock	input	wire logic	Transmit 25/2.5 MHz Clock (from PHY)
pi_tx_clock_en	input	wire logic[3:0]	Enable Transmit 25/2.5 MHz Clock (from PHY)

continues on next page

Table 16 – continued from previous page

Name	Direction	Type	Description
pi_rx_clock	input	wire logic	Receive 125/25/2.5 MHz Clock (from PHY)
pi_rx_clock_en	input	wire logic[2:0]	Enable Receive 125/25/2.5 MHz Clock (from PHY)
pi_loopback	input	wire logic	Operating Mode Information MUX clock domain to TX Clock domain
pi_gigabit	input	wire logic	Clocks divider/MUX information
po_mdio_clock_f	output	wire logic	Output MDIO clock MDIO clock used by the MDIO module
po_host_clock_f	output	wire logic	Output Host Clock Host Free Clock
po_host_clock_g	output	wire logic	Host Gated Clock
po_tx_clock_f	output	wire logic	Output Transmit Clocks Transmit Free 125/25/12.5 MHz Clock for external interface
po_tx_clock_g	output	wire logic[3:0]	po_tx_clock_if , // Transmit Inverted Free 125/25/12.5 MHz Clock for external interface Transmit Gated 125/25/12.5 MHz Clock for external interface
po_rx_clock_f	output	wire logic	Output Receive Clocks Receive Free 125/25/12.5 MHz clock for external interface
po_rx_clock_g	output	wire logic[2:0]	Receive Gated 125/25/12.5 MHz clock for external interface
pi_test_en	input	wire logic	Test and Scan interface signals Test enable (test clock select)
pi_bistmode_en	input	wire logic	

Instances

- *ip_emac_top* > *ip_mac_top_g* > `mac_clk_mng`

Submodules

- **ip_mac_clk_mng_g**
  - gtx_clock_out : *ip_gate_clock_g*
  - host_free_clock : *ip_gate_clock_g*
  - host_gate_clock_1 : *ip_gate_clock_g*
  - host_sreset : *ip_sync_reset_g*
  - hw_host_sreset : *ip_sync_reset_g*
  - mdio_free_clock : *ip_gate_clock_g*
  - mdio_sreset : *ip_sync_reset_g*
  - rx_free_clock : *ip_gate_clock_g*
  - rx_gate_clock_1 : *ip_gate_clock_g*
  - rx_gate_clock_2 : *ip_gate_clock_g*
  - rx_gate_clock_3 : *ip_gate_clock_g*
  - rx_sreset : *ip_sync_reset_g*
  - tx_free_clock : *ip_gate_clock_g*
  - tx_gate_clock_1 : *ip_gate_clock_g*
  - tx_gate_clock_2 : *ip_gate_clock_g*
  - tx_gate_clock_3 : *ip_gate_clock_g*
  - tx_gate_clock_4 : *ip_gate_clock_g*
  - tx_sreset : *ip_sync_reset_g*

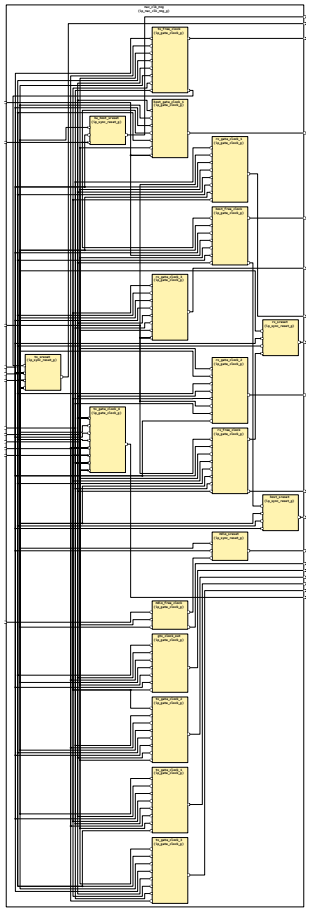


Fig. 16: Flow Diagram of ip_mac_clk_mng_g

6.12 Module ip_mac_dram_001

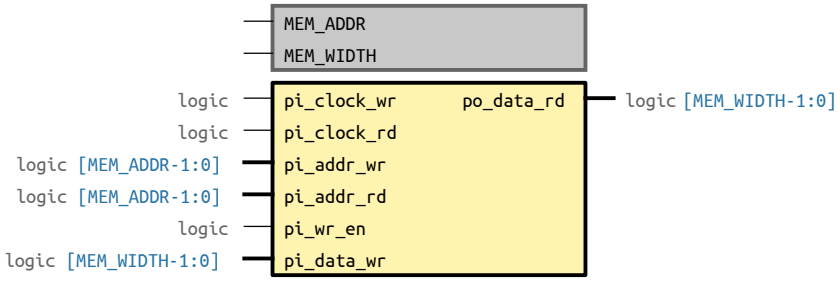


Fig. 17: Block Diagram of ip_mac_dram_001

Overview

The internal DRAM memory modules have the property that when the read address selects a memory location, this location is write protected (locked for write operation). Since when FIFO is empty both read and write addresses points to the same address and the a write operation should be performed, the read memory address should point to an unused memory location in order to unlock the write operation. Moving the internal address to an unused location when write enable modifies the read operation timing since the read DRAM address does not point to the right read address. In order to avoid the timing read response delay a bypass register between input and output data is used. When FIFO is empty the output data reads the bypass register instead of DRAM memory output.

Note: No bypass is necessary for Hash table memory since the read is not performed in the same time memory is writed. Bypass data register and bypass select multiplexer is removed from this module

Table 17: Parameters

Name	Default value	Description
MEM_ADDR	9	Address width (9 -> 512 locations, 10->1024)
MEM_WIDTH	39	Data width

Table 18: Ports

Name	Direction	Type	Description
pi_clock_wr	input	wire logic	Input write clock (multiplexed with scan clock ouside this module)
pi_clock_rd	input	wire logic	Input read clock
pi_addr_wr	input	wire logic[MEM_-ADDR-1:0]	Write address
pi_addr_rd	input	wire logic[MEM_-ADDR-1:0]	Read address
pi_wr_en	input	wire logic	Write enable
pi_data_wr	input	wire logic[MEM_-WIDTH-1:0]	Write data
po_data_rd	output	wire logic[MEM_-WIDTH-1:0]	Read data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_data_dram : ip_mac_dram_001#(.MEM_ADDR(10), .MEM_WIDTH(39))`

Submodules

- **ip_mac_dram_001#(.MEM_ADDR(10), .MEM_WIDTH(39))**
  - dram_001 : *dram_001* #(.MEM_ADDR(10), .MEM_WIDTH(39))

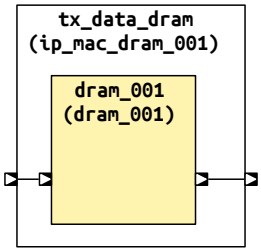


Fig. 18: Flow Diagram of ip_mac_dram_001

6.13 Module ip_mac_dram_002

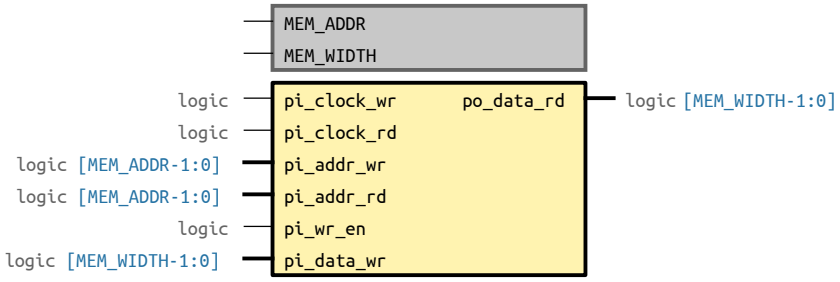


Fig. 19: Block Diagram of ip_mac_dram_002

Overview

The internal DRAM memory modules have the property that when the read address selects a memory location, this location is write protected (locked for write operation). Since when FIFO is empty both read and write addresses points to the same address and the a write operation should be performed, the read memory address should point to an unused memory location in order to unlock the write operation. Moving the internal address to an unused location when write enable modifies the read operation timing since the read DRAM address does not point to the right read address. In order to avoid the timing read response delay a bypass register between input and output data is used. When FIFO is empty the output data reads the bypass register instead of DRAM memory output.

Note: No bypass is necessary for Hash table memory since the read is not performed in the same time memory is writed. Bypass data register and bypass select multiplexer is removed from this module

Table 19: Parameters

Name	Default value	Description
MEM_ADDR	9	Address width (9 -> 512 locations, 10->1024)
MEM_WIDTH	37	Data width

Table 20: Ports

Name	Direction	Type	Description
pi_clock_wr	input	wire logic	Input write clock (multiplexed with scan clock ouside this module)
pi_clock_rd	input	wire logic	Input read clock
pi_addr_wr	input	wire logic[MEM_-ADDR-1:0]	Write address
pi_addr_rd	input	wire logic[MEM_-ADDR-1:0]	Read address
pi_wr_en	input	wire logic	Write enable
pi_data_wr	input	wire logic[MEM_-WIDTH-1:0]	Write data
po_data_rd	output	wire logic[MEM_-WIDTH-1:0]	Read data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_data_dram : ip_mac_dram_002#(.MEM_ADDR(10), .MEM_WIDTH(37))`

Submodules

- **ip_mac_dram_002#(.MEM_ADDR(10), .MEM_WIDTH(37))**
  - dram_001 : *dram_001* #(.MEM_ADDR(10), .MEM_WIDTH(37))

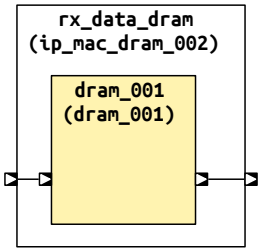


Fig. 20: Flow Diagram of ip_mac_dram_002

6.14 Module ip_mac_dram_003

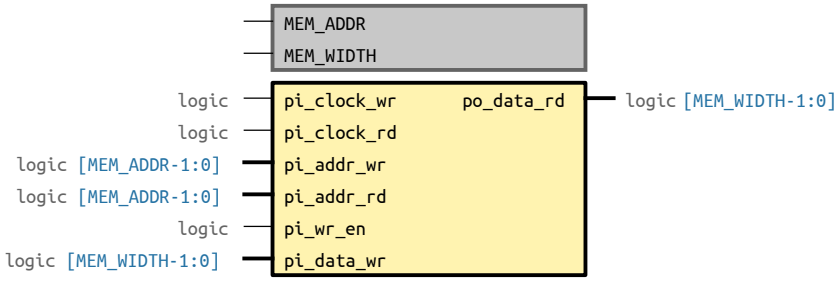


Fig. 21: Block Diagram of ip_mac_dram_003

Overview

The internal DRAM memory modules have the property that when the read address selects a memory location, this location is write protected (locked for write operation). Since when FIFO is empty both read and write addresses points to the same address and the a write operation should be performed, the read memory address should point to an unused memory location in order to unlock the write operation. Moving the internal address to an unused location when write enable modifies the read operation timing since the read DRAM address does not point to the right read address. In order to avoid the timing read response delay a bypass register between input and output data is used. When FIFO is empty the output data reads the bypass register instead of DRAM memory output.

Note: No bypass is necessary for Hash table memory since the read is not performed in the same time memory is writed. Bypass data register and bypass select multiplexer is removed from this module

Table 21: Parameters

Name	Default value	Description
MEM_ADDR	4	Address width (9 -> 512 locations, 10->1024)
MEM_WIDTH	16	Data width

Table 22: Ports

Name	Direction	Type	Description
pi_clock_wr	input	wire logic	Input write clock (multiplexed with scan clock ouside this module)
pi_clock_rd	input	wire logic	Input read clock
pi_addr_wr	input	wire logic[MEM_-ADDR-1:0]	Write address
pi_addr_rd	input	wire logic[MEM_-ADDR-1:0]	Read address
pi_wr_en	input	wire logic	Write enable
pi_data_wr	input	wire logic[MEM_-WIDTH-1:0]	Write data
po_data_rd	output	wire logic[MEM_-WIDTH-1:0]	Read data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_dram_hash_1 : ip_mac_dram_003#(.MEM_ADDR(4), .MEM_WIDTH(16))`

Submodules

- **ip_mac_dram_003#(.MEM_ADDR(4), .MEM_WIDTH(16))**
  - dram_001 : *dram_001* #(.MEM_ADDR(4), .MEM_WIDTH(16))

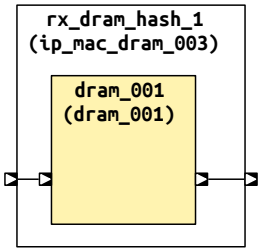


Fig. 22: Flow Diagram of ip_mac_dram_003

6.15 Module ip_mac_dram_004

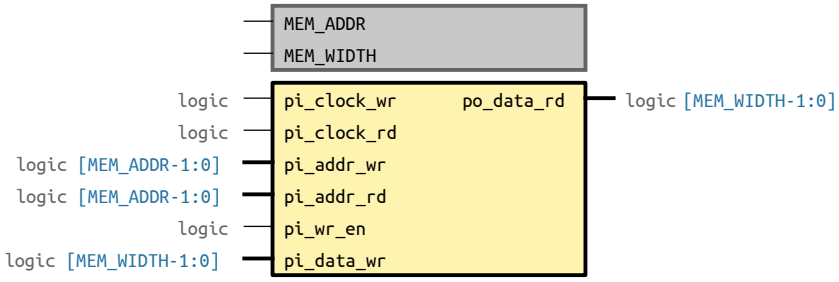


Fig. 23: Block Diagram of ip_mac_dram_004

Overview

The internal DRAM memory modules have the property that when the read address selects a memory location, this location is write protected (locked for write operation). Since when FIFO is empty both read and write addresses points to the same address and the a write operation should be performed, the read memory address should point to an unused memory location in order to unlock the write operation. Moving the internal address to an unused location when write enable modifies the read operation timing since the read DRAM address does not point to the right read address. In order to avoid the timing read response delay a bypass register between input and output data is used. When FIFO is empty the output data reads the bypass register instead of DRAM memory output.

Note: No bypass is necessary for Hash table memory since the read is not performed in the same time memory is writed. Bypass data register and bypass select multiplexer is removed from this module

Table 23: Parameters

Name	Default value	Description
MEM_ADDR	4	Address width (9 -> 512 locations, 10->1024)
MEM_WIDTH	32	Data width

Table 24: Ports

Name	Direction	Type	Description
pi_clock_wr	input	wire logic	Input write clock (multiplexed with scan clock ouside this module)
pi_clock_rd	input	wire logic	Input read clock
pi_addr_wr	input	wire logic[MEM_-ADDR-1:0]	Write address
pi_addr_rd	input	wire logic[MEM_-ADDR-1:0]	Read address
pi_wr_en	input	wire logic	Write enable
pi_data_wr	input	wire logic[MEM_-WIDTH-1:0]	Write data
po_data_rd	output	wire logic[MEM_-WIDTH-1:0]	Read data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_dram_hash_0 : ip_mac_dram_004#(.MEM_ADDR(4), .MEM_WIDTH(32))`

Submodules

- **ip_mac_dram_004#(.MEM_ADDR(4), .MEM_WIDTH(32))**
  - dram_001 : *dram_001* #(.MEM_ADDR(4), .MEM_WIDTH(32))

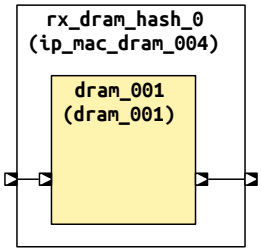


Fig. 24: Flow Diagram of ip_mac_dram_004

6.16 Module ip_mac_fc_dec_g

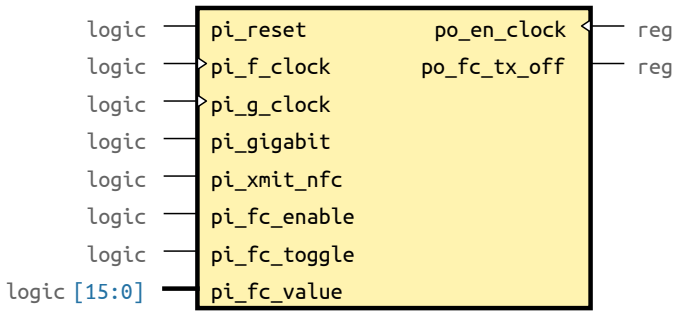


Fig. 25: Block Diagram of ip_mac_fc_dec_g

Table 25: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Software/Hardware Reset (receive clock domain)
pi_f_clock	input	wire logic	Free Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_g_clock	input	wire logic	Gated Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_en_clock	output	var reg	Enable Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_xmit_nfc	input	wire logic	Transmit FSM data frame transmit enable (when full duplex)
pi_fc_enable	input	wire logic	NOTE: This signal is not asserted during flow control frame transmission Receive flow control enable (flow control decoding enable)
pi_fc_toggle	input	wire logic	New pause frame received (pi_fc_value valid)
pi_fc_value	input	wire logic[15:0]	Pause time (from RX EMAC, FC frame decoding)
po_fc_tx_off	output	var reg	Received FC packet (Transmit stop command)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `fc_dec`

6.17 Module ip_mac_fc_gen_g

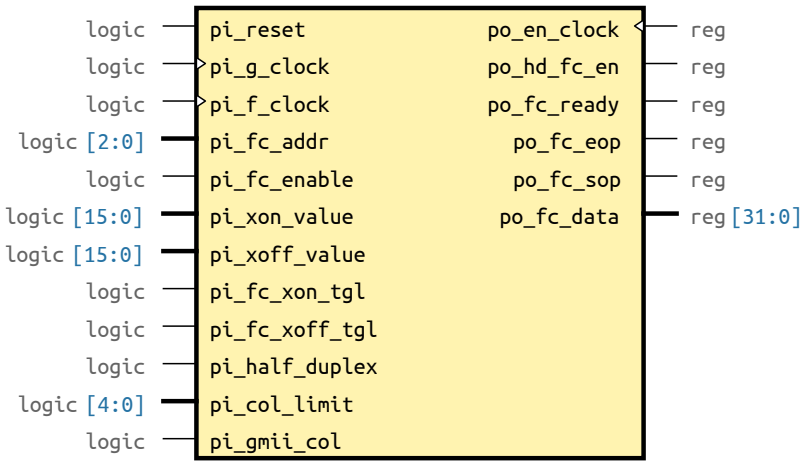


Fig. 26: Block Diagram of ip_mac_fc_gen_g

Overview

The flow control (PAUSE) operation is used to inhibit transmission of data frames for a specified period of time. A MAC Control client wishing to inhibit transmission of data frames from another station on the network generates a MAC CONTROL frame specifying: - The globally assigned 48-bit multicast destination address 01-80-C2-00-00-01H - The PAUSE opcode 00-01H - The CONTROL frame type 88-08H - A request Pause Value (16-bit value) indicating the length of time for which it wishes to inhibit data frame transmission.

The PAUSE operation cannot be used to inhibit transmission of MAC Control frames. PAUSE frames shall only be sent by MAC's configured to the full duplex mode of operation. The globally assigned 48-bit multicast address 01-80-C2-00-00-01 has been reserved for use in MAC Control PAUSE frames for inhibiting transmission of data frames from a MAC in a full duplex mode.

Table 26: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)
pi_g_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, gated clock)
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, free clock)
po_en_clock	output	var reg	Transmit GMII/MII 125/25/2.5 MHz clock gated clock enable
pi_fc_addr	input	wire logic[2:0]	Command interface Next FC word request (from TX state)
pi_fc_enable	input	wire logic	Enable flow control
pi_xon_value	input	wire logic[15:0]	from configuration XOFF flow control pause value
pi_xoff_value	input	wire logic[15:0]	from configuration XON flow control pause value
pi_fc_xon_tgl	input	wire logic	Request insert FC XON/XOFF (each time the following input toggle) from RX EMAC insert XOFF flow control information

continues on next page

Table 26 – continued from previous page

Name	Direction	Type	Description
pi_fc_xoff_tgl	input	wire logic	from RX EMAC insert XON flow control information
pi_half_duplex	input	wire logic	Half duplex flow control Operating in half duplex mode
pi_col_limit	input	wire logic[4:0]	Half duplex back pressure collision limit
pi_gmii_col	input	wire logic	Collision indication used to count the collisions during HD FC enable
po_hd_fc_en	output	var reg	Half Duplex flow control enable
po_fc_ready	output	var reg	FC frame interface Request to insert a new FC frame
po_fc_eop	output	var reg	Note: When asserted has the meaning of ready and start of frame Flow control frame end of frame
po_fc_sop	output	var reg	Flow control frame start of frame
po_fc_data	output	var reg[31:0]	Flow control frame data

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_fc_gen`

6.18 Module ip_mac_host_if

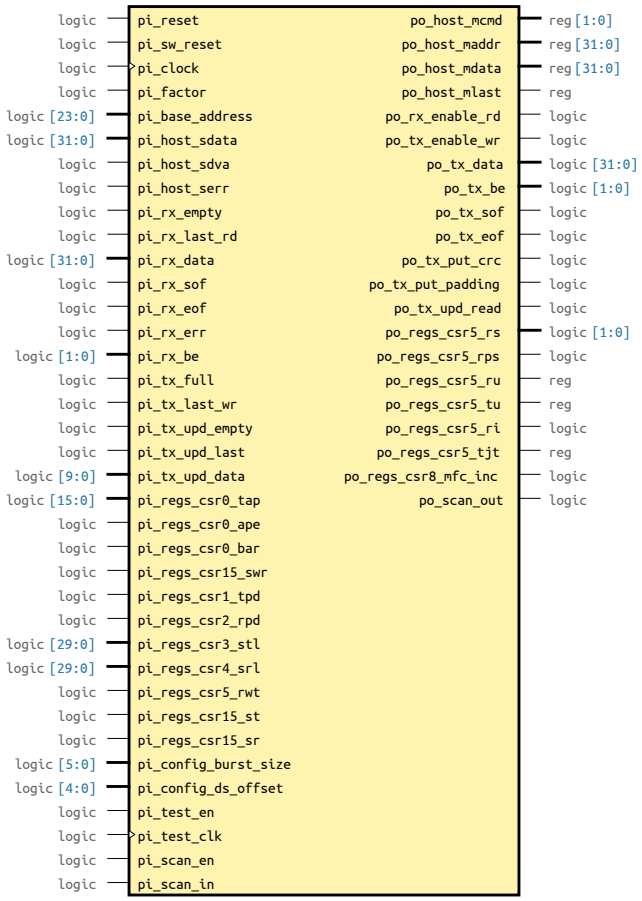


Fig. 27: Block Diagram of ip_mac_host_if

Table 27: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous reset
pi_sw_reset	input	wire logic	software reset
pi_clock	input	wire logic	host clock
pi_factor	input	wire logic	host clock enable
pi_base_address	input	wire logic[23:0]	HOST Interface device int mem space base address
po_host_mcmd	output	var reg[1:0]	command
po_host_maddr	output	var reg[31:0]	address
po_host_mdata	output	var reg[31:0]	data to write
po_host_mlast	output	var reg	last word
pi_host_sdata	input	wire logic[31:0]	data to read
pi_host_sdva	input	wire logic	data valid
pi_host_serr	input	wire logic	error during last transaction
po_rx_enable_rd	output	wire logic	RX FIFO interface rx fifo read command
pi_rx_empty	input	wire logic	rx fifo full
pi_rx_last_rd	input	wire logic	rx fifo almost full
pi_rx_data	input	wire logic[31:0]	rx fifo data
pi_rx_sof	input	wire logic	rx fifo eof

continues on next page

Table 27 – continued from previous page

Name	Direction	Type	Description
pi_rx_eof	input	wire logic	rx fifo eof
pi_rx_err	input	wire logic	rx fifo error
pi_rx_be	input	wire logic[1:0]	rx fifo data byte enable
po_tx_enable_wr	output	wire logic	rx fifo read command
pi_tx_full	input	wire logic	rx fifo full
pi_tx_last_wr	input	wire logic	rx fifo almost full
po_tx_data	output	wire logic[31:0]	rx fifo data
po_tx_be	output	wire logic[1:0]	tx fifo data byte enable
po_tx_sof	output	wire logic	tx fifo start of frame
po_tx_eof	output	wire logic	tx fifo end of frame
po_tx_put_crc	output	wire logic	put crc indication (valid only when sof=1)
po_tx_put_padding	output	wire logic	put padding indication (valid only when sof=1)
po_tx_upd_read	output	wire logic	TX update fifo interface
pi_tx_upd_empty	input	wire logic	
pi_tx_upd_last	input	wire logic	
pi_tx_upd_data	input	wire logic[9:0]	
pi_regs_csr0_tap	input	wire logic[15:0]	Registers bank interface tx automatic polling period CSR0[31:16]
pi_regs_csr0_ape	input	wire logic	tx auto polling enable CSR[15]
pi_regs_csr0_bar	input	wire logic	bus arbitration
pi_regs_csr15_swr	input	wire logic	software reset
pi_regs_csr1_tpd	input	wire logic	transmit poll demand
pi_regs_csr2_rpd	input	wire logic	receive poll demand
pi_regs_csr3_stl	input	wire logic[29:0]	transmit descriptor base address
pi_regs_csr4_srl	input	wire logic[29:0]	receive descriptor base address
po_regs_csr5_rs	output	wire logic[1:0]	receive process state
pi_regs_csr5_rwt	input	wire logic	receive watchdog timeout
po_regs_csr5_rps	output	wire logic	receive process stopped
po_regs_csr5_ru	output	var reg	receive buffer unavailable
po_regs_csr5_tu	output	var reg	transmit buffer unavailable
po_regs_csr5_ri	output	wire logic	receive interupt
po_regs_csr5_tjt	output	var reg	signals Transmit jabber timeout error to the CSR5 register; this will signal to the HOST to perform a software reset to the EMAC
pi_regs_csr15_st	input	wire logic	start/stop transmit
pi_regs_csr15_sr	input	wire logic	start / stop receive
po_regs_csr8_-mfc_inc	output	wire logic	missed frame counter increment command
pi_config_burst_-size	input	wire logic[5:0]	limit for the rx,tx burst transfers
pi_config_ds_offset	input	wire logic[4:0]	pi_rx_mac_ds_poll, //from the RX EMAC part (new frame arrived, or new data arrived) offset to increment the address if a descriptor
pi_test_en	input	wire logic	Test and Scan interface signals Test enable
pi_test_clk	input	wire logic	Test clock
pi_scan_en	input	wire logic	SCAN chain shift enable
pi_scan_in	input	wire logic	SCAN chain input
po_scan_out	output	wire logic	SCAN chain output

Always Blocks

- always @(posedge clock or negedge reset)

manages the next descriptor aquire when rx_ds_needed is asserted, asserts rx_ds_req when pi_host_arb_ds_dv is asserted load pi_host_arb_ds_data into current_ds_addr and deasserts po_host_arb_ds_req the one that asserted ds_needed waits for pi_host_arb_ds_dv to reset

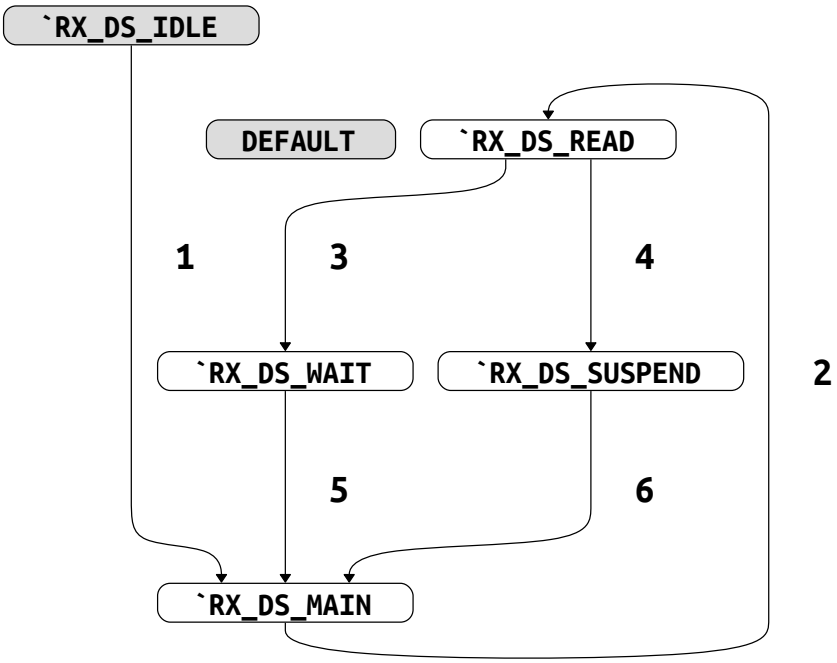


Table 28: FSM Transitions for rx_ds_state

#	Current State	Next State	Condition
1	`RX_DS_IDLE	`RX_DS_MAIN	[!(! pi_regs_csr15_sr)]
2	`RX_DS_MAIN	`RX_DS_READ	[(rx_ds_allowed)]
3	`RX_DS_READ	`RX_DS_WAIT	[((pi_host_sdva) && (pi_host_sdata[31]))]
4	`RX_DS_READ	`RX_DS_SUS-PEND	[((pi_host_sdva) && !(pi_host_sdata[31]))]
5	`RX_DS_WAIT	`RX_DS_MAIN	[(rx_valid_ds_read)]
6	`RX_DS_SUS-PEND	`RX_DS_MAIN	[(rx_mac_ds_poll ‖ pi_regs_csr2_rpd)]

- always @(posedge clock or negedge reset)

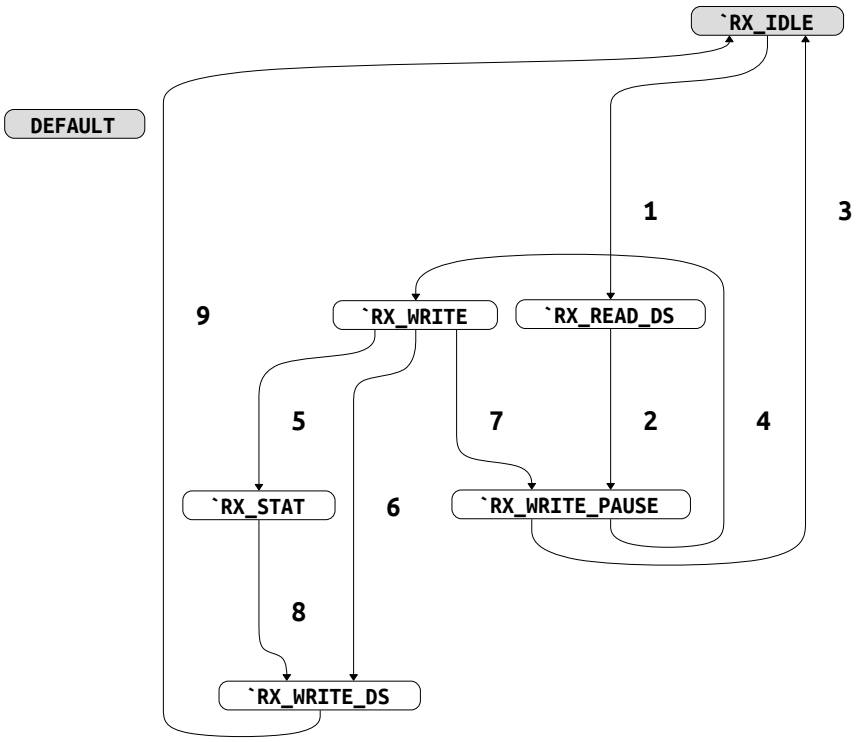


Table 29: FSM Transitions for rx_state

#	Current State	Next State	Condition
1	`RX_IDLE	`RX_READ_DS	[(rx_allowed)]
2	`RX_READ_DS	`RX_WRITE_-PAUSE	[((pi_host_sdva) && !(rx_burst_cnt == 0) && !(rx_-burst_cnt == 1) && (rx_burst_cnt == 2))]
3	`RX_WRITE_-PAUSE	`RX_IDLE	[(! rx_ds_valid)]
4	`RX_WRITE_-PAUSE	`RX_WRITE	[(!(! rx_ds_valid) && !(! pi_rx_empty && ! pi_rx_-last_rd && ! rx_req) && (rx_allowed))]
5	`RX_WRITE	`RX_STAT	[(!(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && (rx_frame_complete))]
6	`RX_WRITE	`RX_WRITE_DS	[(!(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && (rx_buff_-complete) && !(! rx_cur_buff && ! rx_ds1[24] && rx_ds1[21 : 11] != 0))]
7	`RX_WRITE	`RX_WRITE_-PAUSE	[(!(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && !(rx_-buff_complete) && (rx_burst_complete)), (!(rx_-first_word_in_burst) && !(! pi_host_sdva) && (rx_-mlast) && !(rx_frame_complete) && !(rx_buff_com-plete) && !(rx_burst_complete)), (!(rx_first_word_-in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && (rx_buff_complete) && (! rx_cur_buff && ! rx_ds1[24] && rx_ds1[21 : 11] != 0))]
8	`RX_STAT	`RX_WRITE_DS	[(! pi_rx_empty)]
9	`RX_WRITE_DS	`RX_IDLE	[(rx_allowed && pi_host_sdva)]

- always @(posedge clock or negedge reset)

manages the next descriptor aquire

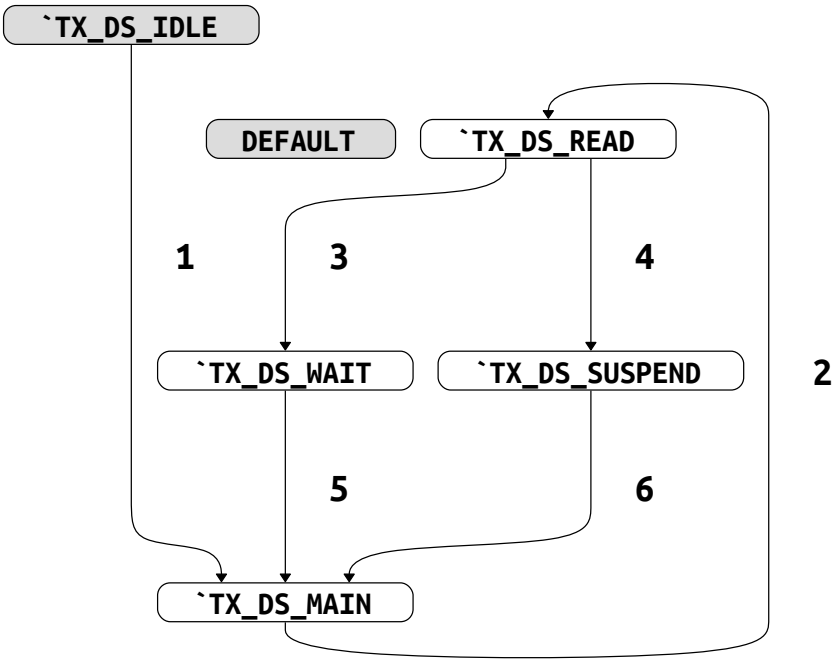


Table 30: FSM Transitions for tx_ds_state

#	Current State	Next State	Condition
1	`TX_DS_IDLE	`TX_DS_MAIN	[!(! pi_regs_csr15_st)]
2	`TX_DS_MAIN	`TX_DS_READ	[(tx_ds_allowed)]
3	`TX_DS_READ	`TX_DS_WAIT	[((pi_host_sdva) && (pi_host_sdata[31]))]
4	`TX_DS_READ	`TX_DS_SUS-PEND	[((pi_host_sdva) && !(pi_host_sdata[31]))]
5	`TX_DS_WAIT	`TX_DS_MAIN	[(tx_valid_ds_read)]
6	`TX_DS_SUS-PEND	`TX_DS_MAIN	[(pi_regs_csr0_ape && tx_ds_poll_cnt == pi_regs_-csr0_tap ‖ pi_regs_csr1_tpd)]

- always @(posedge clock or negedge reset)

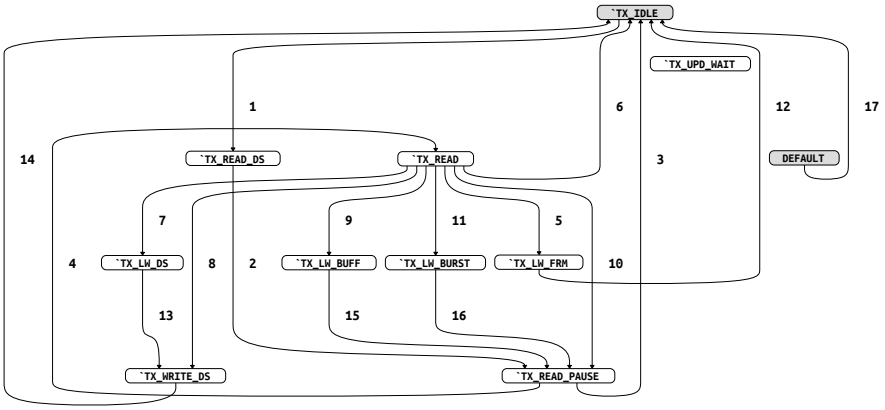


Table 31: FSM Transitions for tx_state

#	Current State	Next State	Condition
1	`TX_IDLE	`TX_READ_DS	[(!(! pi_regs_csr15_st) && !(tx_ds_addr_valid && ! pi_tx_full && ! pi_tx_last_wr && ! tx_upd_fifo_full && ! tx_req) && (tx_allowed))]
2	`TX_READ_DS	`TX_READ_-PAUSE	[((pi_host_sdva) && !(tx_burst_cnt == 0) && !(tx_-burst_cnt == 1) && (tx_burst_cnt == 2))]
3	`TX_READ_-PAUSE	`TX_IDLE	[(! tx_ds_valid)]
4	`TX_READ_-PAUSE	`TX_READ	[(!(! tx_ds_valid) && !(! pi_tx_full && ! pi_tx_last_-wr && ! tx_req) && (tx_allowed))]
5	`TX_READ	`TX_LW_FRM	[((pi_host_sdva) && (tx_mlast1) && (tx_word_cnt >= 16'h1000) && (pi_tx_last_wr)), ((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_-buff_cnt == tx_curr_buff_size - 1) && (tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && (tx_ds1[30]) && (pi_tx_last_wr))]
6	`TX_READ	`TX_IDLE	[((pi_host_sdva) && (tx_mlast1) && (tx_word_cnt >= 16'h1000) && !(pi_tx_last_wr)), ((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_-buff_cnt == tx_curr_buff_size - 1) && (tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && (tx_ds1[30]) && !(pi_tx_last_wr))]
7	`TX_READ	`TX_LW_DS	[((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_buff_cnt == tx_curr_buff_size - 1) && (tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && !(tx_ds1[30]) && (pi_tx_last_wr))]
8	`TX_READ	`TX_WRITE_DS	[((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_buff_cnt == tx_curr_buff_size - 1) && (tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && !(tx_ds1[30]) && !(pi_tx_last_wr))]
9	`TX_READ	`TX_LW_BUFF	[((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_buff_cnt == tx_curr_buff_size - 1) && !(tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && (pi_tx_last_wr))]
10	`TX_READ	`TX_READ_-PAUSE	[((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && !(tx_buff_cnt == tx_curr_buff_-size - 1) && (tx_burst_cnt == pi_config_burst_size - 1) && !(pi_tx_last_wr)), ((pi_host_sdva) && (tx_-mlast1) && !(tx_word_cnt >= 16'h1000) && (tx_-buff_cnt == tx_curr_buff_size - 1) && !(tx_cur_buff || tx_ds1[24] || tx_ds1[21 : 11] == 0) && !(pi_tx_last_-wr))]
11	`TX_READ	`TX_LW_BURST	[((pi_host_sdva) && !(tx_mlast1) && (pi_tx_last_wr || pi_tx_full)), ((pi_host_sdva) && (tx_mlast1) && !(tx_word_cnt >= 16'h1000) && !(tx_buff_cnt == tx_-curr_buff_size - 1) && (tx_burst_cnt == pi_config_-burst_size - 1) && (pi_tx_last_wr))]
12	`TX_LW_FRM	`TX_IDLE	[!(pi_tx_last_wr)]
13	`TX_LW_DS	`TX_WRITE_DS	[!(pi_tx_last_wr)]
14	`TX_WRITE_DS	`TX_IDLE	[(pi_host_sdva)]
15	`TX_LW_BUFF	`TX_READ_-PAUSE	[!(pi_tx_last_wr)]
16	`TX_LW_BURST	`TX_READ_-PAUSE	[!(pi_tx_last_wr)]
17	default	`TX_IDLE	[EMPTY]

- always @(posedge clock or negedge reset)

this process reads from both tx_upd_fifo and tx_upd_resp_fifo, constructs TX_RDS0 and assert tx_ds_req to arbiter along with address(tx_upd_fifo) and data (tx_upd_resp_fifo)

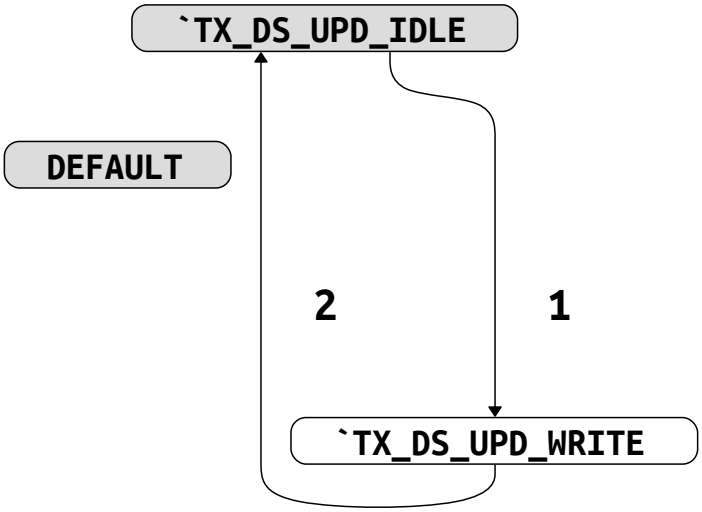


Table 32: FSM Transitions for tx_upd_state

#	Current State	Next State	Condition
1	`TX_DS_UPD_-IDLE	`TX_DS_UPD_-WRITE	[(tx_upd_req && tx_upd_allowed)]
2	`TX_DS_UPD_-WRITE	`TX_DS_UPD_-IDLE	[(pi_host_sdva)]

- always @(arb_state or rx_allowed or tx_allowed or rx_ds_allowed or tx_ds_allowed or tx_upd_allowed)

arbiter next state calculation

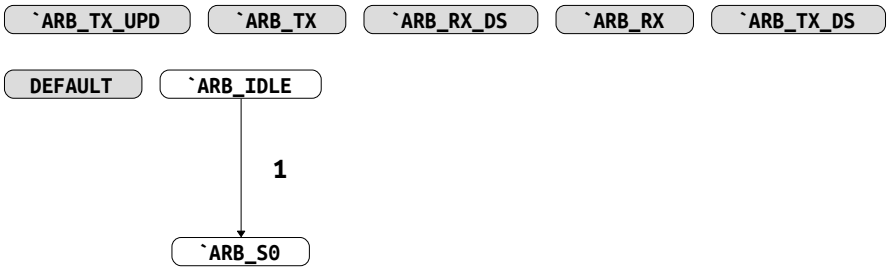


Table 33: FSM Transitions for arb_state

#	Current State	Next State	Condition
1	`ARB_IDLE	`ARB_S0	[(arb_enable)]

6.19 Module ip_mac_hostif_arb

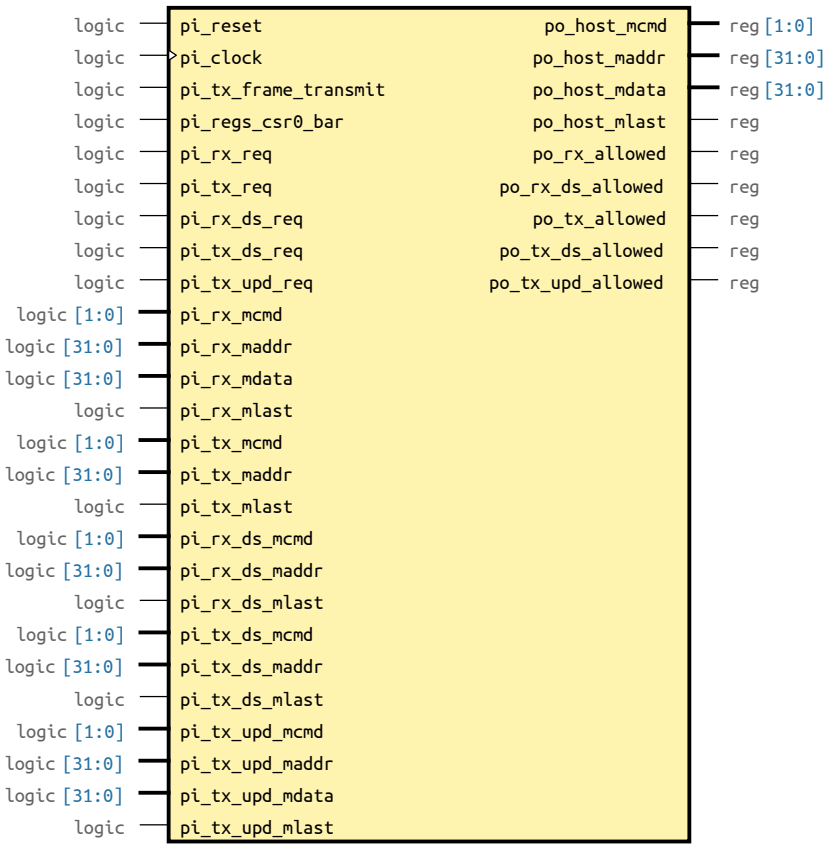


Fig. 28: Block Diagram of ip_mac_hostif_arb

Table 34: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous pi_reset
pi_clock	input	wire logic	host pi_clock
pi_tx_frame_transmit	input	wire logic	from pi_mac_hostif_tx
pi_regs_csr0_bar	input	wire logic	from banks regs
pi_rx_req	input	wire logic	
pi_tx_req	input	wire logic	
pi_rx_ds_req	input	wire logic	
pi_tx_ds_req	input	wire logic	
pi_tx_upd_req	input	wire logic	
pi_rx_mcmd	input	wire logic[1:0]	
pi_rx_maddr	input	wire logic[31:0]	
pi_rx_mdata	input	wire logic[31:0]	
pi_rx_mlast	input	wire logic	
pi_tx_mcmd	input	wire logic[1:0]	
pi_tx_maddr	input	wire logic[31:0]	
pi_tx_mlast	input	wire logic	pi_tx_mdata,
pi_rx_ds_mcmd	input	wire logic[1:0]	
pi_rx_ds_maddr	input	wire logic[31:0]	

continues on next page

Table 34 – continued from previous page

Name	Direction	Type	Description
pi_rx_ds_mlast	input	wire logic	
pi_tx_ds_mcmd	input	wire logic[1:0]	
pi_tx_ds_maddr	input	wire logic[31:0]	
pi_tx_ds_mlast	input	wire logic	
pi_tx_upd_mcmd	input	wire logic[1:0]	
pi_tx_upd_maddr	input	wire logic[31:0]	
pi_tx_upd_mdata	input	wire logic[31:0]	
pi_tx_upd_mlast	input	wire logic	
po_host_mcmd	output	var reg[1:0]	
po_host_maddr	output	var reg[31:0]	
po_host_mdata	output	var reg[31:0]	
po_host_mlast	output	var reg	
po_rx_allowed	output	var reg	
po_rx_ds_allowed	output	var reg	
po_tx_allowed	output	var reg	
po_tx_ds_allowed	output	var reg	
po_tx_upd_allowed	output	var reg	

Always Blocks

- always @(arb_state or po_rx_allowed or po_tx_allowed or po_rx_ds_allowed or po_tx_ds_allowed or po_-tx_upd_allowed)

arbiter next state calculation

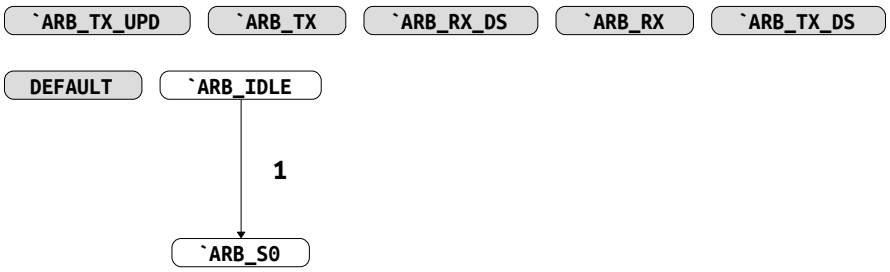


Table 35: FSM Transitions for arb_state

#	Current State	Next State	Condition
1	`ARB_IDLE	`ARB_S0	[!(~ pi_reset)]

Instances

- *ip_emac_top* > *ip_mac_hostif_top* > `hostif_arb`

6.20 Module ip_mac_hostif_rx

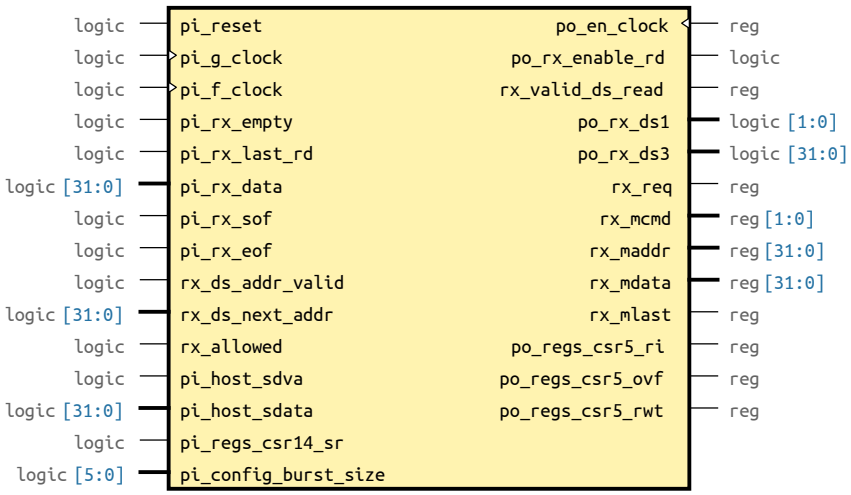


Fig. 29: Block Diagram of ip_mac_hostif_rx

Table 36: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous pi_reset
pi_g_clock	input	wire logic	host gated clock
pi_f_clock	input	wire logic	host free clock
po_en_clock	output	var reg	enable clock condition
po_rx_enable_rd	output	wire logic	RX FIFO interface rx fifo read command
pi_rx_empty	input	wire logic	rx fifo full
pi_rx_last_rd	input	wire logic	rx fifo almost full
pi_rx_data	input	wire logic[31:0]	rx fifo data
pi_rx_sof	input	wire logic	rx fifo eof
pi_rx_eof	input	wire logic	rx fifo eof
rx_ds_addr_valid	input	wire logic	ip_mac_hostif_rxds from ip_mac_hostif_rxds (currently fetched descriptor address valid)
rx_ds_next_addr	input	wire logic[31:0]	from ip_mac_hostif_rxds (currently fetched descriptor address)
rx_valid_ds_read	output	var reg	
po_rx_ds1	output	wire logic[1:0]	mapped to rx, rx_ds1[25:24]
po_rx_ds3	output	wire logic[31:0]	
rx_req	output	var reg	arbiter
rx_allowed	input	wire logic	
rx_mcmd	output	var reg[1:0]	
rx_maddr	output	var reg[31:0]	
rx_mdata	output	var reg[31:0]	
rx_mlast	output	var reg	
pi_host_sdva	input	wire logic	host interface
pi_host_sdata	input	wire logic[31:0]	
pi_regs_csr14_sr	input	wire logic	ip_mac_regs_bank (config)
pi_config_burst_size	input	wire logic[5:0]	limit for the rx,tx burst transfers
po_regs_csr5_ri	output	var reg	

continues on next page

Table 36 – continued from previous page

Name	Direction	Type	Description
po_regs_csr5_ovf	output	var reg	
po_regs_csr5_rwt	output	var reg	

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Gated Clock Enable

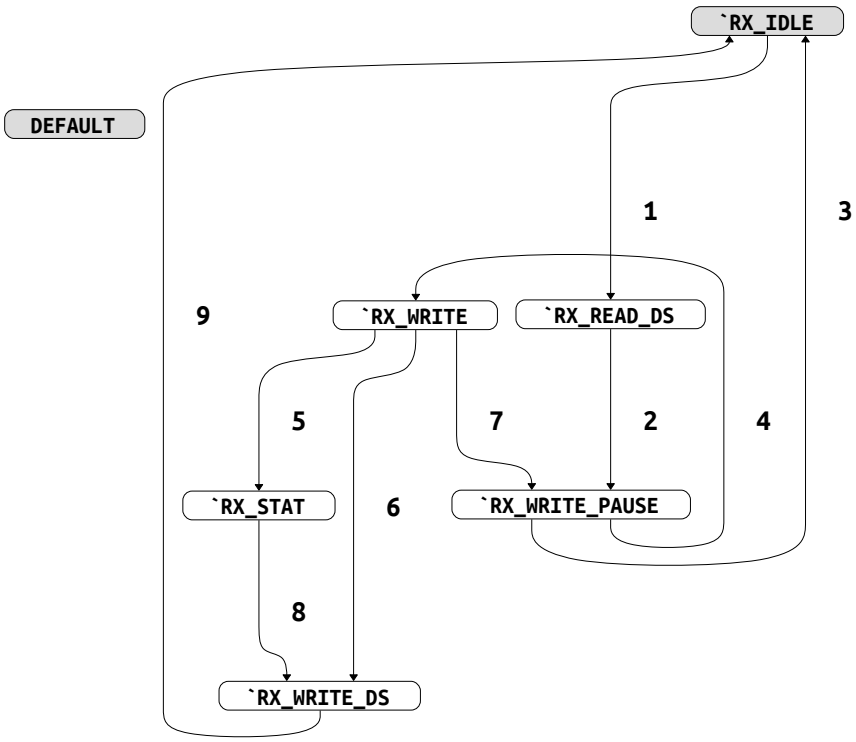


Table 37: FSM Transitions for rx_state

#	Current State	Next State	Condition
1	`RX_IDLE	`RX_READ_DS	[(!(~ pi_reset) && !(! rx_ds_addr_valid) && !(rx_-ds_addr_valid && ! rx_req) && (rx_allowed))]
2	`RX_READ_DS	`RX_WRITE_-PAUSE	[(!(~ pi_reset) && (pi_host_sdva) && !(rx_burst_cnt == 0) && !(rx_burst_cnt == 1) && (rx_burst_cnt == 2))]
3	`RX_WRITE_-PAUSE	`RX_IDLE	[(!(~ pi_reset) && (! rx_ds_valid))]
4	`RX_WRITE_-PAUSE	`RX_WRITE	[(!(~ pi_reset) && !(! rx_ds_valid) && !(! pi_rx_-empty && ! pi_rx_last_rd && ! rx_req) && (rx_-allowed))]
5	`RX_WRITE	`RX_STAT	[(!(~ pi_reset) && !(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && (rx_frame_com-plete))]

continues on next page

Table 37 – continued from previous page

#	Current State	Next State	Condition
6	`RX_WRITE	`RX_WRITE_DS	[(!(~ pi_reset) && !(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && (rx_buff_complete) && !(! rx_cur_buff && ! rx_ds1[24] && rx_ds1[21 : 11] != 0))]
7	`RX_WRITE	`RX_WRITE_PAUSE	[(!(~ pi_reset) && !(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && !(rx_buff_complete) && (rx_burst_complete)), (!(~ pi_reset) && !(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && !(rx_buff_complete) && !(rx_burst_complete)), (!(~ pi_reset) && !(rx_first_word_in_burst) && !(! pi_host_sdva) && (rx_mlast) && !(rx_frame_complete) && (rx_buff_complete) && (! rx_cur_buff && ! rx_ds1[24] && rx_ds1[21 : 11] != 0))]
8	`RX_STAT	`RX_WRITE_DS	[(!(~ pi_reset) && (! pi_rx_empty))]
9	`RX_WRITE_DS	`RX_IDLE	[(!(~ pi_reset) && (rx_allowed && pi_host_sdva))]

Instances

- *ip_emac_top* > *ip_mac_hostif_top* > `hostif_rx`

6.21 Module ip_mac_hostif_rxds

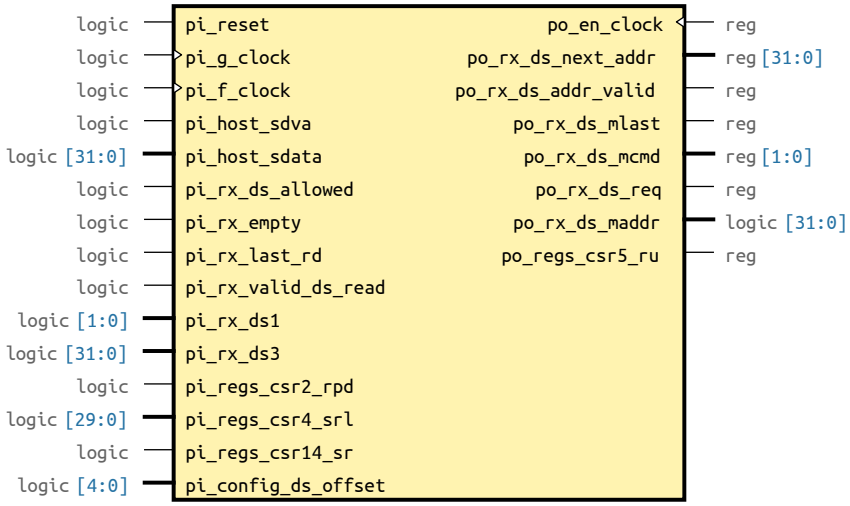


Fig. 30: Block Diagram of ip_mac_hostif_rxds

Table 38: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous pi_reset
pi_g_clock	input	wire logic	host gated clock
pi_f_clock	input	wire logic	host free clock
po_en_clock	output	var reg	enable clock condition
pi_host_sdva	input	wire logic	
pi_host_sdata	input	wire logic[31:0]	mapped to pi_host_sdata[31]
pi_rx_ds_allowed	input	wire logic	
pi_rx_empty	input	wire logic	
pi_rx_last_rd	input	wire logic	
po_rx_ds_next_-addr	output	var reg[31:0]	output
po_rx_ds_addr_-valid	output	var reg	output
po_rx_ds_mlast	output	var reg	output
po_rx_ds_mcmd	output	var reg[1:0]	output
po_rx_ds_req	output	var reg	output
po_rx_ds_maddr	output	wire logic[31:0]	output
pi_rx_valid_ds_-read	input	wire logic	input
pi_rx_ds1	input	wire logic[1:0]	holds current rx descriptor body mapped to rx, pi_rx_ds1[25:24]
pi_rx_ds3	input	wire logic[31:0]	
pi_regs_csr2_rpd	input	wire logic	Registers bank interface receive poll demand
pi_regs_csr4_srl	input	wire logic[29:0]	receive descriptor base address
po_regs_csr5_ru	output	var reg	receive buffer unavailable
pi_regs_csr14_sr	input	wire logic	start / stop receive
pi_config_ds_offset	input	wire logic[4:0]	offset to increment the address if a descriptor

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Gated Clock Enable

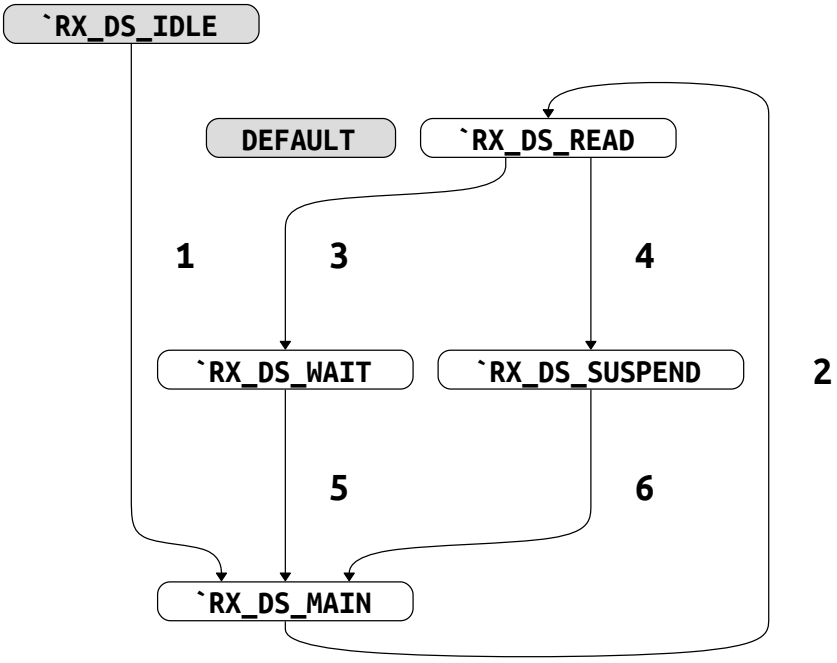


Table 39: FSM Transitions for rx_ds_state

#	Current State	Next State	Condition
1	`RX_DS_IDLE	`RX_DS_MAIN	[(!(~ pi_reset) && !(! pi_regs_csr14_sr))]
2	`RX_DS_MAIN	`RX_DS_READ	[(!(~ pi_reset) && (pi_rx_ds_allowed))]
3	`RX_DS_READ	`RX_DS_WAIT	[(!(~ pi_reset) && (pi_host_sdva) && (pi_host_-sdata))]
4	`RX_DS_READ	`RX_DS_SUS-PEND	[(!(~ pi_reset) && (pi_host_sdva) && !(pi_host_-sdata))]
5	`RX_DS_WAIT	`RX_DS_MAIN	[(!(~ pi_reset) && (pi_rx_valid_ds_read))]
6	`RX_DS_SUS-PEND	`RX_DS_MAIN	[(!(~ pi_reset) && (rx_mac_ds_poll ‖ pi_regs_csr2_-rpd))]

Instances

- *ip_emac_top* > *ip_mac_hostif_top* > `hostif_rxds`

6.22 Module ip_mac_hostif_top

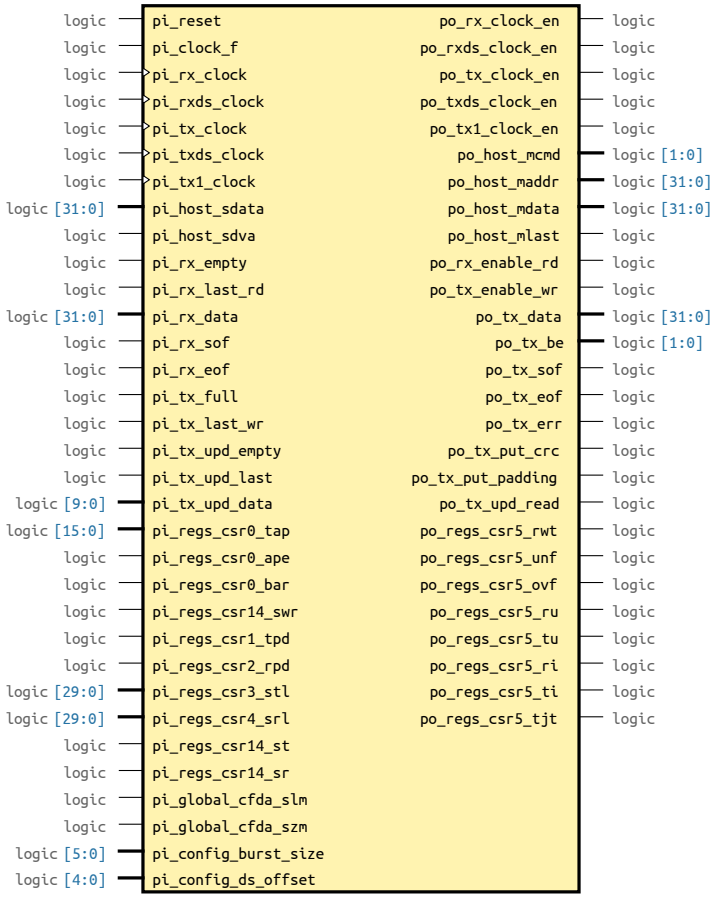


Fig. 31: Block Diagram of ip_mac_hostif_top

Table 40: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	global asynchronous reset
pi_clock_f	input	wire logic	host clock
pi_rx_clock	input	wire logic	
pi_rxds_clock	input	wire logic	
pi_tx_clock	input	wire logic	
pi_txds_clock	input	wire logic	
pi_tx1_clock	input	wire logic	
po_rx_clock_en	output	wire logic	clock enable for gated clocks
po_rxds_clock_en	output	wire logic	
po_tx_clock_en	output	wire logic	
po_txds_clock_en	output	wire logic	
po_tx1_clock_en	output	wire logic	
po_host_mcmd	output	wire logic[1:0]	command
po_host_maddr	output	wire logic[31:0]	address
po_host_mdata	output	wire logic[31:0]	data to write
po_host_mlast	output	wire logic	last word
pi_host_sdata	input	wire logic[31:0]	data to read
pi_host_sdva	input	wire logic	data valid

continues on next page

Table 40 – continued from previous page

Name	Direction	Type	Description
po_rx_enable_rd	output	wire logic	rx fifo read command
pi_rx_empty	input	wire logic	rx fifo full
pi_rx_last_rd	input	wire logic	rx fifo almost full
pi_rx_data	input	wire logic[31:0]	rx fifo data
pi_rx_sof	input	wire logic	rx fifo eof
pi_rx_eof	input	wire logic	rx fifo eof
po_tx_enable_wr	output	wire logic	rx fifo read command
pi_tx_full	input	wire logic	rx fifo full
pi_tx_last_wr	input	wire logic	rx fifo almost full
po_tx_data	output	wire logic[31:0]	rx fifo data
po_tx_be	output	wire logic[1:0]	tx fifo data byte enable
po_tx_sof	output	wire logic	tx fifo start of frame
po_tx_eof	output	wire logic	tx fifo end of frame
po_tx_err	output	wire logic	tx fifo error
po_tx_put_crc	output	wire logic	put crc indication (valid only when sof=1)
po_tx_put_padding	output	wire logic	put padding indication (valid only when sof=1)
po_tx_upd_read	output	wire logic	
pi_tx_upd_empty	input	wire logic	
pi_tx_upd_last	input	wire logic	
pi_tx_upd_data	input	wire logic[9:0]	
pi_regs_csr0_tap	input	wire logic[15:0]	tx automatic polling period CSR0[31:16]
pi_regs_csr0_ape	input	wire logic	tx auto polling enable CSR[15]
pi_regs_csr0_bar	input	wire logic	bus arbitration
pi_regs_csr14_swr	input	wire logic	software reset
pi_regs_csr1_tpd	input	wire logic	transmit poll demand
pi_regs_csr2_rpd	input	wire logic	receive poll demand
pi_regs_csr3_stl	input	wire logic[29:0]	transmit descriptor base address
pi_regs_csr4_srl	input	wire logic[29:0]	receive descriptor base address
po_regs_csr5_rwt	output	wire logic	receive watchdog timeout
po_regs_csr5_unf	output	wire logic	transmit underflow
po_regs_csr5_ovf	output	wire logic	receive overflow
po_regs_csr5_ru	output	wire logic	receive buffer unavailable
po_regs_csr5_tu	output	wire logic	transmit buffer unavailable
po_regs_csr5_ri	output	wire logic	receive interupt
po_regs_csr5_ti	output	wire logic	transmit interupt
po_regs_csr5_tjt	output	wire logic	signals Transmit jabber timeout error to the CSR5 register; this will signal to the HOST to perform a software reset to the EMAC
pi_regs_csr14_st	input	wire logic	start/stop transmit
pi_regs_csr14_sr	input	wire logic	start / stop receive
pi_global_cfda_slm	input	wire logic	sleep mode enable
pi_global_cfda_-szm	input	wire logic	snooze mode enable
pi_config_burst_-size	input	wire logic[5:0]	limit for the rx,tx burst transfers
pi_config_ds_offset	input	wire logic[4:0]	offset to increment the address if a descriptor

Instances

- *ip_emac_top* > `host_if`

Submodules

- **ip_mac_hostif_top**
  - hostif_arb : *ip_mac_hostif_arb*
  - hostif_rx : *ip_mac_hostif_rx*
  - hostif_rxds : *ip_mac_hostif_rxds*
  - hostif_tx : *ip_mac_hostif_tx*
  - hostif_txds : *ip_mac_hostif_txds*

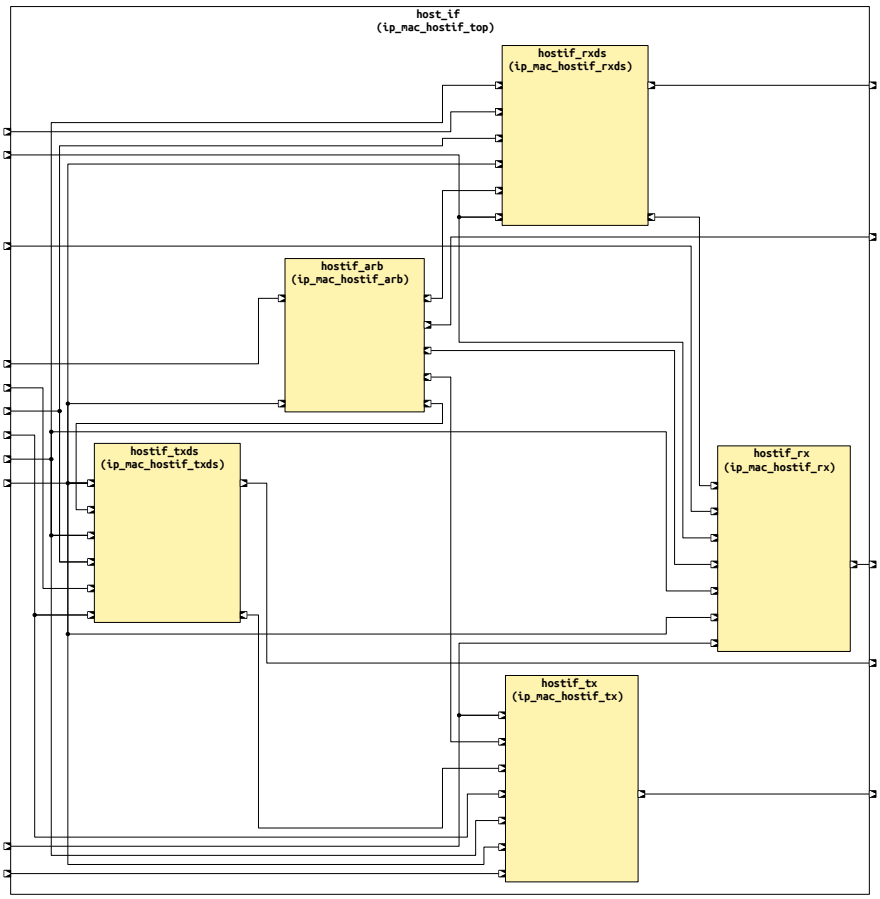


Fig. 32: Flow Diagram of ip_mac_hostif_top

6.23 Module ip_mac_hostif_tx

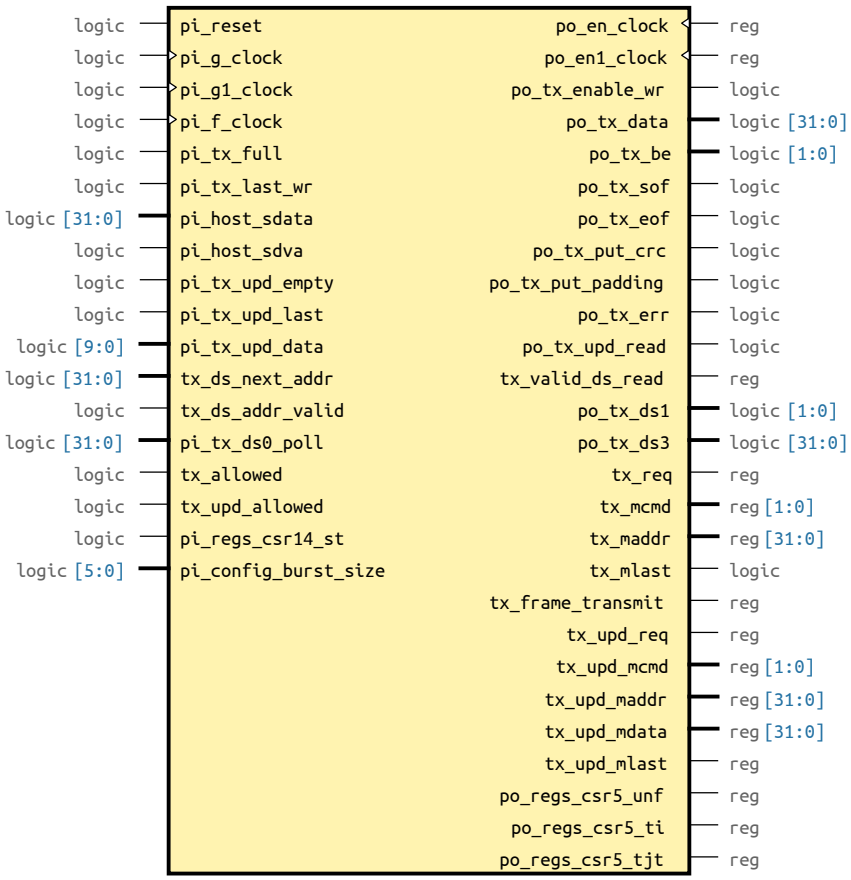


Fig. 33: Block Diagram of ip_mac_hostif_tx

Table 41: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous pi_reset
pi_g_clock	input	wire logic	host gated clock
pi_g1_clock	input	wire logic	host gated clock 1
pi_f_clock	input	wire logic	host free clock
po_en_clock	output	var reg	enable clock condition
po_en1_clock	output	var reg	enable clock condition 1
po_tx_enable_wr	output	wire logic	tx mac fifo interface rx fifo read command
pi_tx_full	input	wire logic	rx fifo full
pi_tx_last_wr	input	wire logic	rx fifo almost full
po_tx_data	output	wire logic[31:0]	rx fifo data
po_tx_be	output	wire logic[1:0]	tx fifo data byte enable
po_tx_sof	output	wire logic	tx fifo start of frame
po_tx_eof	output	wire logic	tx fifo end of frame
po_tx_put_crc	output	wire logic	put crc indication (valid only when sof=1)
po_tx_put_padding	output	wire logic	put padding indication (valid only when sof=1)
po_tx_err	output	wire logic	put error indication (valid only when eof=1)
pi_host_sdata	input	wire logic[31:0]	host interface from host interface

continues on next page

Table 41 – continued from previous page

Name	Direction	Type	Description
pi_host_sdva	input	wire logic	from host interface
po_tx_upd_read	output	wire logic	TX update fifo interface
pi_tx_upd_empty	input	wire logic	
pi_tx_upd_last	input	wire logic	
pi_tx_upd_data	input	wire logic[9:0]	
tx_valid_ds_read	output	var reg	ip_mac_hostif_txds to ip_mac_hostif_txds (new descriptor polling enabled)
tx_ds_next_addr	input	wire logic[31:0]	from ip_mac_hostif_txds (new descriptor address)
tx_ds_addr_valid	input	wire logic	from ip_mac_hostif_txds (new descriptor address valid)
pi_tx_ds0_poll	input	wire logic[31:0]	from ip_mac_hostif_txds (new descriptor address valid)
po_tx_ds1	output	wire logic[1:0]	
po_tx_ds3	output	wire logic[31:0]	
tx_allowed	input	wire logic	ip_mac_hostif_arb from pi_mac_hostif_-arb (tx upd req allowed)
tx_req	output	var reg	request to arb from tx upd ds process
tx_mcmd	output	var reg[1:0]	tx update ds mcmd
tx_maddr	output	var reg[31:0]	tx update ds maddr
tx_mlast	output	wire logic	tx update ds mlast
tx_frame_transmit	output	var reg	to ip_mac_hostif_arb (used in arbitration process)
tx_upd_allowed	input	wire logic	from pi_mac_hostif_arb (tx upd req allowed)
tx_upd_req	output	var reg	request to arb from tx upd ds process
tx_upd_mcmd	output	var reg[1:0]	tx update ds mcmd
tx_upd_maddr	output	var reg[31:0]	tx update ds maddr
tx_upd_mdata	output	var reg[31:0]	tx update ds mdata
tx_upd_mlast	output	var reg	tx update ds mlast
po_regs_csr5_unf	output	var reg	ip_mac_regs_bank (config) transmit underflow
po_regs_csr5_ti	output	var reg	transmit frame complete (to CSR5 TI)
po_regs_csr5_tjt	output	var reg	transmit jabber timeout error
pi_regs_csr14_st	input	wire logic	start/stop transmit
pi_config_burst_-size	input	wire logic[5:0]	limit for rx tx burst transfer

Always Blocks

- always @(posedge pi_g1_clock or negedge pi_reset)

this process reads from both tx_upd_fifo and tx_upd_resp_fifo, constructs TX_RDS0 and assert tx_ds_req to arbiter along with address(tx_upd_fifo) and data (tx_upd_resp_fifo)

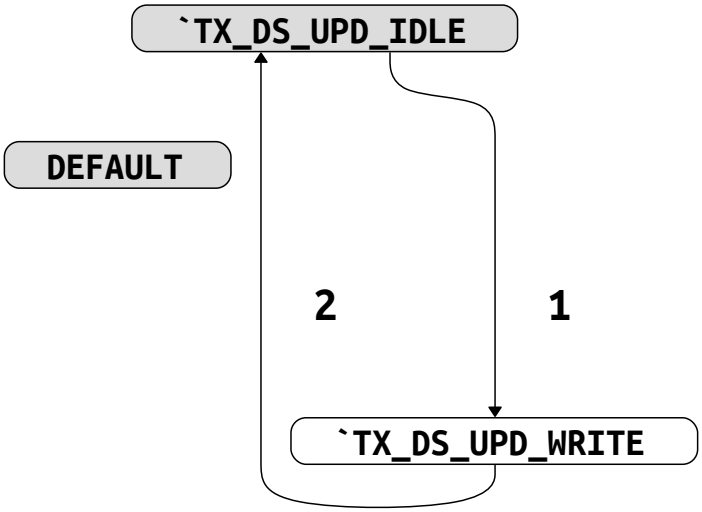


Table 42: FSM Transitions for tx_upd_state

#	Current State	Next State	Condition
1	`TX_DS_UPD_-IDLE	`TX_DS_UPD_-WRITE	[(!(~ pi_reset) && (tx_upd_req && tx_upd_allowed))]
2	`TX_DS_UPD_-WRITE	`TX_DS_UPD_-IDLE	[(!(~ pi_reset) && (pi_host_sdva))]

- always @(posedge pi_f_clock or negedge pi_reset)

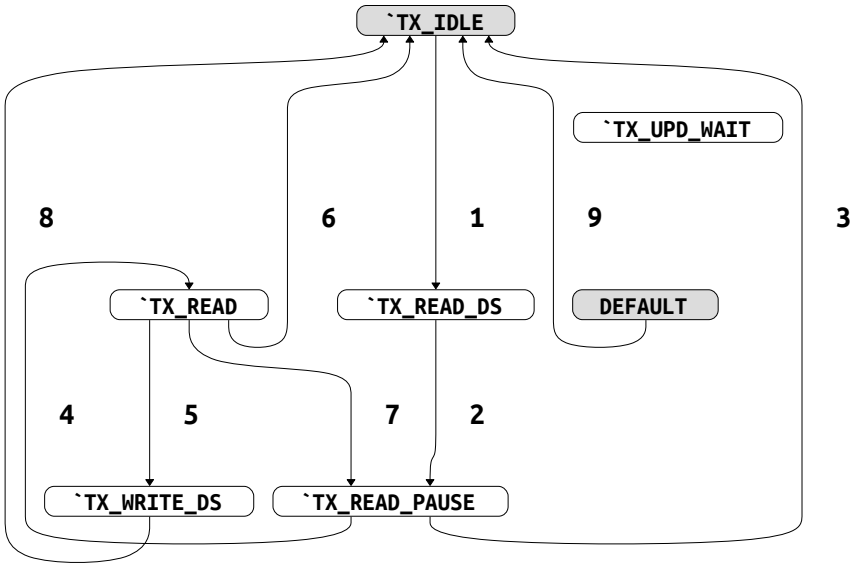


Table 43: FSM Transitions for tx_state

#	Current State	Next State	Condition
1	`TX_IDLE	`TX_READ_DS	[(!(~ pi_reset) && !(! pi_regs_csr14_st ‖ ! tx_ds_-addr_valid) && !(tx_ds_addr_valid && ! pi_tx_full && ! pi_tx_last_wr && ! tx_upd_fifo_full && ! tx_-req) && (tx_allowed))]

continues on next page

Table 43 – continued from previous page

#	Current State	Next State	Condition
2	`TX_READ_DS	`TX_READ_-PAUSE	[(!(~ pi_reset) && (pi_host_sdva) && !(tx_burst_cnt == 0) && !(tx_burst_cnt == 1) && (tx_burst_cnt == 2))]
3	`TX_READ_-PAUSE	`TX_IDLE	[(!(~ pi_reset) && (! tx_ds_valid))]
4	`TX_READ_-PAUSE	`TX_READ	[(!(~ pi_reset) && !(! tx_ds_valid) && !(! pi_tx_full && ! pi_tx_last_wr && ! tx_req) && (tx_allowed))]
5	`TX_READ	`TX_WRITE_DS	[(!(~ pi_reset) && !(tx_allowed && pi_host_sdva) && (! tx_mlast && tx_word_remained && ! pi_tx_last_-wr && ! pi_tx_full) && (tx_next_state == 4'b0100)), (!(~ pi_reset) && (tx_allowed && pi_host_sdva) && (tx_mlast1) && (tx_buff_cnt == tx_curr_buff_size - 1) && (tx_cur_buff ‖ tx_ds1[24] ‖ tx_ds1[21 : 11] == 0) && !(tx_ds1[30]) && !(tx_word_remained))]
6	`TX_READ	`TX_IDLE	[(!(~ pi_reset) && !(tx_allowed && pi_host_sdva) && (! tx_mlast && tx_word_remained && ! pi_tx_last_-wr && ! pi_tx_full) && !(tx_next_state == 4'b0100) && (tx_next_state == 4'b0000)), (!(~ pi_reset) && (tx_allowed && pi_host_sdva) && (tx_mlast1) && (tx_buff_cnt == tx_curr_buff_size - 1) && (tx_cur_-buff ‖ tx_ds1[24] ‖ tx_ds1[21 : 11] == 0) && (tx_-ds1[30]) && !(tx_word_remained))]
7	`TX_READ	`TX_READ_-PAUSE	[(!(~ pi_reset) && !(tx_allowed && pi_host_sdva) && (! tx_mlast && tx_word_remained && ! pi_-tx_last_wr && ! pi_tx_full) && !(tx_next_state == 4'b0100) && !(tx_next_state == 4'b0000)), (!(~ pi_-reset) && (tx_allowed && pi_host_sdva) && (tx_-mlast1) && !(tx_buff_cnt == tx_curr_buff_size - 1) && (tx_burst_cnt == pi_config_burst_size - 1) && !(tx_word_remained)), (!(~ pi_reset) && (tx_allowed && pi_host_sdva) && (tx_mlast1) && (tx_buff_cnt == tx_curr_buff_size - 1) && !(tx_cur_buff ‖ tx_-ds1[24] ‖ tx_ds1[21 : 11] == 0) && !(tx_word_re-mained))]
8	`TX_WRITE_DS	`TX_IDLE	[(!(~ pi_reset) && (tx_allowed && pi_host_sdva))]
9	default	`TX_IDLE	[!(~ pi_reset)]

Instances

- *ip_emac_top* > *ip_mac_hostif_top* > `hostif_tx`

6.24 Module ip_mac_hostif_txds

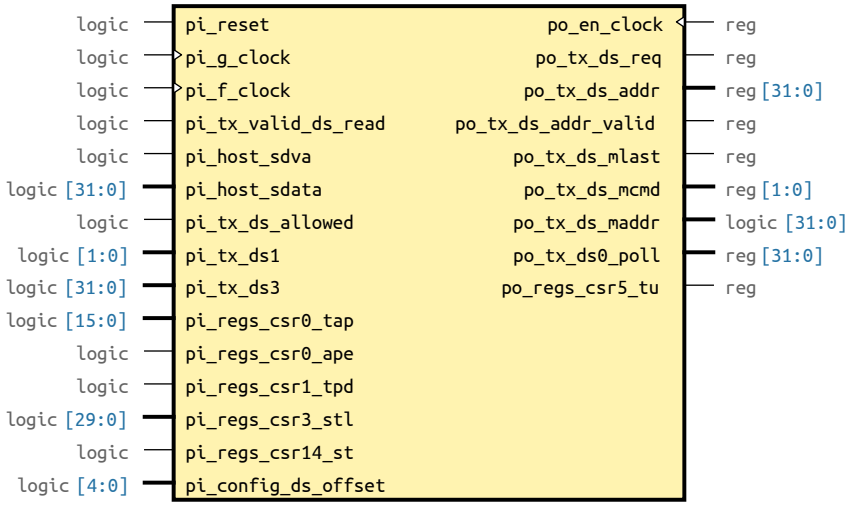


Fig. 34: Block Diagram of ip_mac_hostif_txds

Table 44: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global interface global asynchronous pi_reset
pi_g_clock	input	wire logic	host pi_g_clock
pi_f_clock	input	wire logic	host pi_g_clock
po_en_clock	output	var reg	enable clock condition
po_tx_ds_req	output	var reg	output
po_tx_ds_addr	output	var reg[31:0]	output , mapped to old tx_ds_next_addr
po_tx_ds_addr_-valid	output	var reg	output
po_tx_ds_mlast	output	var reg	output
po_tx_ds_mcmd	output	var reg[1:0]	output
po_tx_ds_maddr	output	wire logic[31:0]	output
pi_tx_valid_ds_-read	input	wire logic	from pi_mac_hostif_tx (signals that new ds polling is enabled)
pi_host_sdva	input	wire logic	
pi_host_sdata	input	wire logic[31:0]	
pi_tx_ds_allowed	input	wire logic	pi_host_serr, //from host interface
po_tx_ds0_poll	output	var reg[31:0]	
pi_tx_ds1	input	wire logic[1:0]	mapped to tx pi_tx_ds1[25:24]
pi_tx_ds3	input	wire logic[31:0]	
pi_regs_csr0_tap	input	wire logic[15:0]	Registers bank interface tx automatic polling period CSR0[31:16]
pi_regs_csr0_ape	input	wire logic	tx auto polling enable CSR[15]
pi_regs_csr1_tpd	input	wire logic	transmit poll demand
pi_regs_csr3_stl	input	wire logic[29:0]	transmit descriptor base address
po_regs_csr5_tu	output	var reg	transmit buffer unavailable
pi_regs_csr14_st	input	wire logic	start/stop transmit
pi_config_ds_offset	input	wire logic[4:0]	offset to increment the address if a descriptor

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

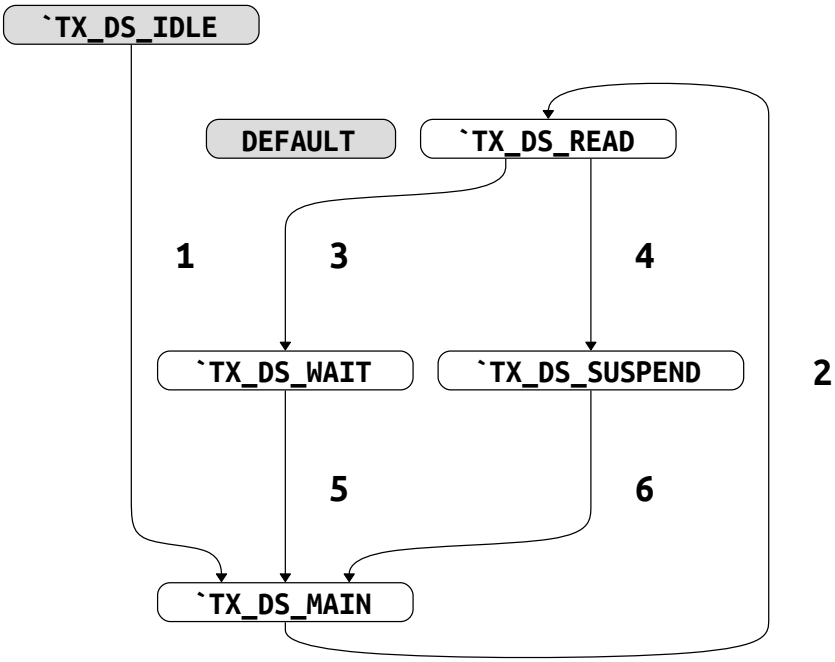


Table 45: FSM Transitions for tx_ds_state

#	Current State	Next State	Condition
1	`TX_DS_IDLE	`TX_DS_MAIN	[(!(~ pi_reset) && !(! pi_regs_csr14_st))]
2	`TX_DS_MAIN	`TX_DS_READ	[(!(~ pi_reset) && (pi_tx_ds_allowed))]
3	`TX_DS_READ	`TX_DS_WAIT	[(!(~ pi_reset) && (pi_host_sdva) && (pi_host_-sdata[31]))]
4	`TX_DS_READ	`TX_DS_SUS-PEND	[(!(~ pi_reset) && (pi_host_sdva) && !(pi_host_-sdata[31]))]
5	`TX_DS_WAIT	`TX_DS_MAIN	[(!(~ pi_reset) && (pi_tx_valid_ds_read))]
6	`TX_DS_SUS-PEND	`TX_DS_MAIN	[(!(~ pi_reset) && (pi_regs_csr0_ape && tx_ds_poll_-cnt == pi_regs_csr0_tap || pi_regs_csr1_tpd))]

Instances

- *ip_emac_top* > *ip_mac_hostif_top* > `hostif_txds`

6.25 Module ip_mac_mdio_g

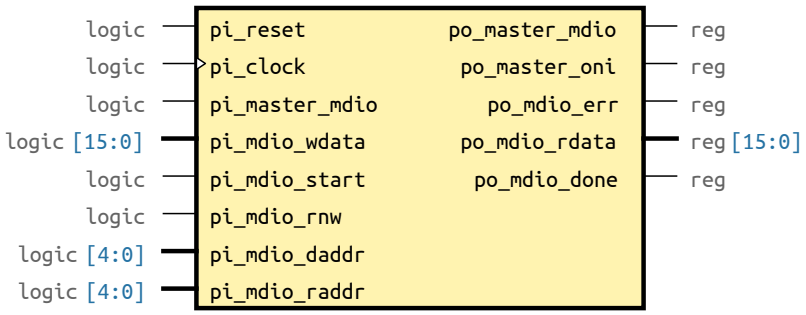


Fig. 35: Block Diagram of ip_mac_mdio_g

Overview

The EMAC has a master management interface, which is used to access the PHY configuration registers. Depending on the configuration, the EMAC will either read the values from the PHY registers (auto-configuration mode) or set the values according to the programmed values (programming mode). The MDIO (po_master_mdio/pi_master_mdio) and MDC (po_master_mdc) ports are used to serially write and read management interface registers. A 2.5 MHz clock waveform must be provided to the MDC port on this interface.

Every management read/write instruction frame contains the following fields:

- PRE, Preamble - The EMAC management interface generates a full 32-bit preamble
- ST, Start of Frame (A "01" pattern indicates the start of frame)
- OP, Operation Code - A read instruction is indicated by "01", while a write instruction is indicated by "10"
- DEVAD, Device Address - A five-bit device address follows the opcode, with the most significant bit transmitted first.
- REGAD, Register Address - A five-bit register address follows the DEVAD field, with the most significant bit transmitted first. This field is used to access the management registers of the PHY.
- TA, Turnaround - The next two bit times are used to avoid contention on the MDIO port during a read transaction. For a read transaction, the PHY device and the system MDIO should be in tri-state for the first cycle of turnaround. The PHY device drives zero during the second cycle of the turnaround. For write transactions, the EMAC should drive 1 during the first cycle of the turnaround and zero during the second cycle.
- Data - The last 16 bits of the frame are the actual data bits. For a write operation, these bits are sent to the PHY device. For a read operation, the PHY device drives these bits. In both cases, the most significant bit is transmitted first.
- Idle - This indicates a high-impedance state of the MDIO line. The default state of the MDIO signal is high impedance with a pull-up resistor.

Table 46: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global reset (asynchronous reset)
pi_clock	input	wire logic	Master MDIO reference clock
pi_master_mdio	input	wire logic	MDIO interface Master MDIO data input line (tri-state buffer outside of the module)
po_master_mdio	output	var reg	Master MDIO data output line (tri-state buffer outside of the module)
po_master_oni	output	var reg	Master MDIO direction tri-state buffer control (1 - Output, 0 - Input)

continues on next page

Table 46 – continued from previous page

Name	Direction	Type	Description
po_mdio_err	output	var reg	Configuration interface (HOST clock domain) Indicates that a read from MDIO interface is invalid and the
pi_mdio_wdata	input	wire logic[15:0]	operation should be retried. This is indicated during a read turn-around cycle when the MDIO slave does not drive the MDIO signal to the low state. MDIO write data
po_mdio_rdata	output	var reg[15:0]	MDIO read data
pi_mdio_start	input	wire logic	Setting this bit initiates an MDIO read/write operation. Start indication
po_mdio_done	output	var reg	should remain asserted till the transaction complete. (asynchronous) MDIO transaction complete
pi_mdio_rnw	input	wire logic	This bit indicates the direction of the MDIO operation type:
pi_mdio_daddr	input	wire logic[4:0]	0 - MDIO Write operation, 1 - MDIO read operation This field is used to specify the device address to be accessed.
pi_mdio_raddr	input	wire logic[4:0]	This field is used to specify the register address to be accessed.

Always Blocks

- always @(posedge pi_clock or negedge pi_reset)

MDIO State Machine

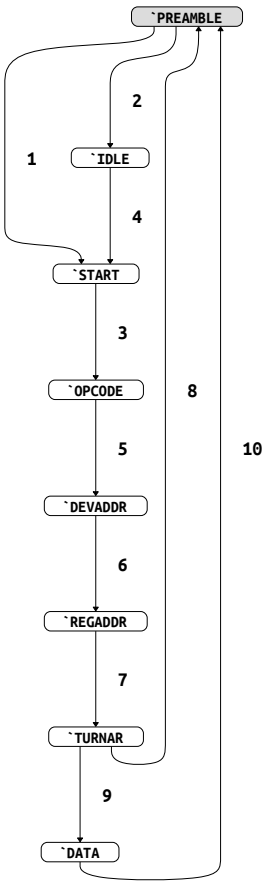


Table 47: FSM Transitions for mdio_state

#	Current State	Next State	Condition
1	`PREAMBLE	`START	[(!(~ pi_reset) && (counter == 5'd0 && mdio_start == 1'b1))]
2	`PREAMBLE	`IDLE	[(!(~ pi_reset) && (counter == 5'd0))]
3	`START	`OPCODE	[(!(~ pi_reset) && (counter == 5'd0))]
4	`IDLE	`START	[(!(~ pi_reset) && (mdio_start == 1'b1))]
5	`OPCODE	`DEVADDR	[(!(~ pi_reset) && (counter == 5'd0))]
6	`DEVADDR	`REGADDR	[(!(~ pi_reset) && (counter == 5'd31))]
7	`REGADDR	`TURNAR	[(!(~ pi_reset) && (counter == 5'd31))]
8	`TURNAR	`PREAMBLE	[(!(~ pi_reset) && (counter == 5'd0 && pi_master_-mdio != 1'b0 && pi_mdio_rnw == 1'b1))]
9	`TURNAR	`DATA	[(!(~ pi_reset) && !(counter == 5'd0 && pi_master_-mdio != 1'b0 && pi_mdio_rnw == 1'b1) && (counter == 5'd0))]
10	`DATA	`PREAMBLE	[(!(~ pi_reset) && (counter == 5'd31))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > `mac_mdio`

6.26 Module ip_mac_regs_bank

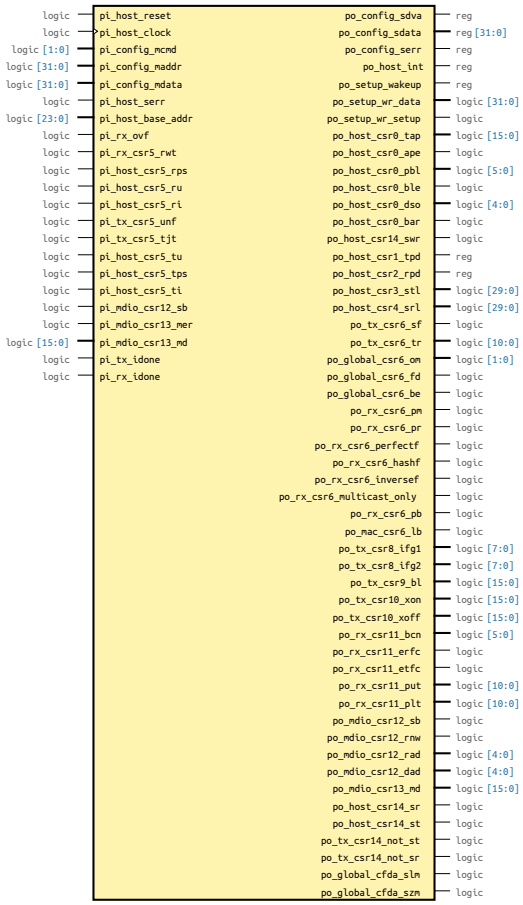


Fig. 36: Block Diagram of ip_mac_regs_bank

Table 48: Ports

Name	Direction	Type	Description
pi_host_reset	input	wire logic	general interface host domain reset
pi_host_clock	input	wire logic	host domain clock
pi_config_mcmd	input	wire logic[1:0]	host access interface
pi_config_maddr	input	wire logic[31:0]	
pi_config_mdata	input	wire logic[31:0]	
po_config_sdva	output	var reg	pi_config_mlast,
po_config_sdata	output	var reg[31:0]	
po_config_serr	output	var reg	
pi_host_serr	input	wire logic	failure on host data interface
pi_host_base_addr	input	wire logic[23:0]	internal space base address
po_host_int	output	var reg	general interrupt to the host
po_setup_wakeup	output	var reg	setup interface
po_setup_wr_data	output	wire logic[31:0]	
po_setup_wr_setup	output	wire logic	
po_host_csr0_tap	output	wire logic[15:0]	host domain register outputs & inputs
po_host_csr0_ape	output	wire logic	
po_host_csr0_pbl	output	wire logic[5:0]	
po_host_csr0_ble	output	wire logic	

continues on next page

Table 48 – continued from previous page

Name	Direction	Type	Description
po_host_csr0_dso	output	wire logic[4:0]	
po_host_csr0_bar	output	wire logic	
po_host_csr14_swr	output	wire logic	
po_host_csr1_tpd	output	var reg	transmit poll demand
po_host_csr2_rpd	output	var reg	receive poll demand
po_host_csr3_stl	output	wire logic[29:0]	transmit descriptor base address
po_host_csr4_srl	output	wire logic[29:0]	receive descriptor base address
pi_rx_ovf	input	wire logic	reveive overflow (from HOST If Rx statistic compilation)
pi_rx_csr5_rwt	input	wire logic	pi_host_csr5_ts, //transmit process state pi_host_csr5_rs, //receive process state receive watchdog timeout(16k limit)
pi_host_csr5_rps	input	wire logic	receive process stopped
pi_host_csr5_ru	input	wire logic	receive buffer unavailable
pi_host_csr5_ri	input	wire logic	receive interupt
pi_tx_csr5_unf	input	wire logic	transmit underflow (from HOST If Tx statistic compilation)
pi_tx_csr5_tjt	input	wire logic	transmit jabber time-out
pi_host_csr5_tu	input	wire logic	transmit buffer unavailable
pi_host_csr5_tps	input	wire logic	transmit process stopped
pi_host_csr5_ti	input	wire logic	transmit interrupt
po_tx_csr6_sf	output	wire logic	transmission store and forward OR transmit when threshold csr6[24:14] reached
po_tx_csr6_tr	output	wire logic[10:0]	transmission threshold
po_global_csr6_om	output	wire logic[1:0]	global operating mode (10/100/1G)
po_global_csr6_fd	output	wire logic	full-duplex mode enable
po_global_csr6_be	output	wire logic	burst enable (when 1G mode selected)
po_rx_csr6_pm	output	wire logic	receive pass all multicast enable
po_rx_csr6_pr	output	wire logic	receive promiscuous mode enable
po_rx_csr6_perfectf	output	wire logic	perfect filtering
po_rx_csr6_hashf	output	wire logic	hash filtering
po_rx_csr6_inversef	output	wire logic	inverse filtering
po_rx_csr6_multicast_only	output	wire logic	imperfect filtering
po_rx_csr6_pb	output	wire logic	receive pass bad frames
po_mac_csr6_lb	output	wire logic	loopback mode enable
po_tx_csr8_ifg1	output	wire logic[7:0]	pi_rx_csr8_mfc_inc, // missed frame counter increment command IFG1
po_tx_csr8_ifg2	output	wire logic[7:0]	IFG2
po_tx_csr9_bl	output	wire logic[15:0]	burst length
po_tx_csr10_xon	output	wire logic[15:0]	tx pause xon
po_tx_csr10_xoff	output	wire logic[15:0]	tx pause xoff
po_rx_csr11_bcn	output	wire logic[5:0]	receive backpressure collision number
po_rx_csr11_erfc	output	wire logic	receive enable flow control
po_rx_csr11_etfc	output	wire logic	transmit enable flow control
po_rx_csr11_put	output	wire logic[10:0]	pause upper threshold
po_rx_csr11_plt	output	wire logic[10:0]	pause lower threshold
po_mdio_csr12_sb	output	wire logic	MDIO start
pi_mdio_csr12_sb	input	wire logic	mdio busy
po_mdio_csr12_rnw	output	wire logic	MDIO r/w

continues on next page

Table 48 – continued from previous page

Name	Direction	Type	Description
po_mdio_csr12_-rad	output	wire logic[4:0]	MDIO register address
po_mdio_csr12_-dad	output	wire logic[4:0]	MDIO device address
pi_mdio_csr13_-mer	input	wire logic	mdio error
po_mdio_csr13_-md	output	wire logic[15:0]	MDIO write data to MDIO if
pi_mdio_csr13_md	input	wire logic[15:0]	MDIO read data from MDIO if
po_host_csr14_sr	output	wire logic	start / stop receive
po_host_csr14_st	output	wire logic	start / stop transmit
po_tx_csr14_not_st	output	wire logic	tx stop transmit
po_tx_csr14_not_-sr	output	wire logic	rx stop receive
pi_tx_idone	input	wire logic	MAC tx initialization done
pi_rx_idone	input	wire logic	MAC rx initialization done
po_global_cfda_-slm	output	wire logic	sleep mode enable
po_global_cfda_-szm	output	wire logic	snooze mode enable

Always Blocks

- always @(negedge pi_host_reset or posedge pi_host_clock)

R/W registers

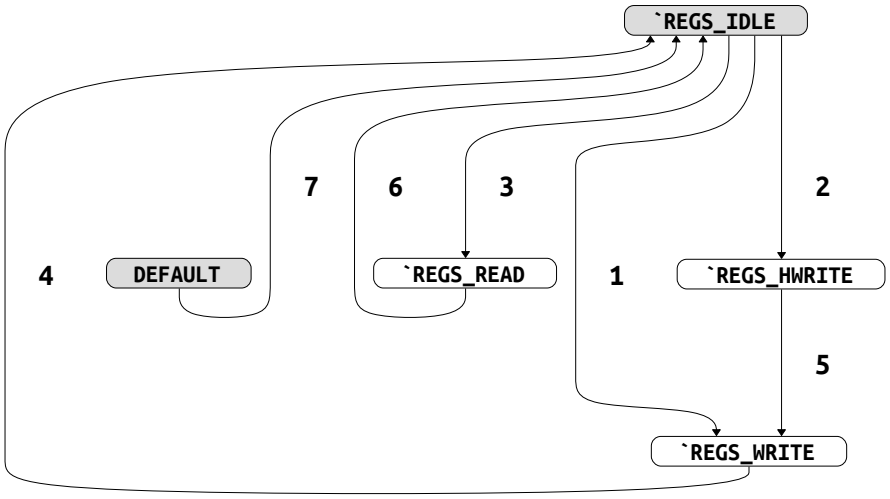


Table 49: FSM Transitions for config_state

#	Current State	Next State	Condition
1	`REGS_IDLE	`REGS_WRITE	[(!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h00)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h04)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h08)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h0c)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h10)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h14)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h18)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h1c)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h20)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h24)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h28)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h2c)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h30)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h34)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h38)), (!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h3c)), (!(~ pi_host_reset) && (device_selected) && (pi_con-fig_mcmd == 2'b01) && (rw_addr == 7'h40)), (!(~ pi_host_reset) && (device_selected) && (pi_config_-mcmd == 2'b01) && (rw_addr == 7'h44))]
2	`REGS_IDLE	`REGS_HWRITE	[(!(~ pi_host_reset) && (device_selected) && (pi_-config_mcmd == 2'b01) && (rw_addr == 7'h48))]
3	`REGS_IDLE	`REGS_READ	[(!(~ pi_host_reset) && (device_selected) && !(pi_-config_mcmd == 2'b01) && (pi_config_mcmd == 2'b10))]
4	`REGS_WRITE	`REGS_IDLE	[!(~ pi_host_reset)]
5	`REGS_HWRITE	`REGS_WRITE	[(!(~ pi_host_reset) && (hash_cnt == 1))]
6	`REGS_READ	`REGS_IDLE	[!(~ pi_host_reset)]
7	default	`REGS_IDLE	[!(~ pi_host_reset)]

Instances

- *ip_emac_top* > `regs_bank`

6.27 Module ip_mac_rx_fifo_g

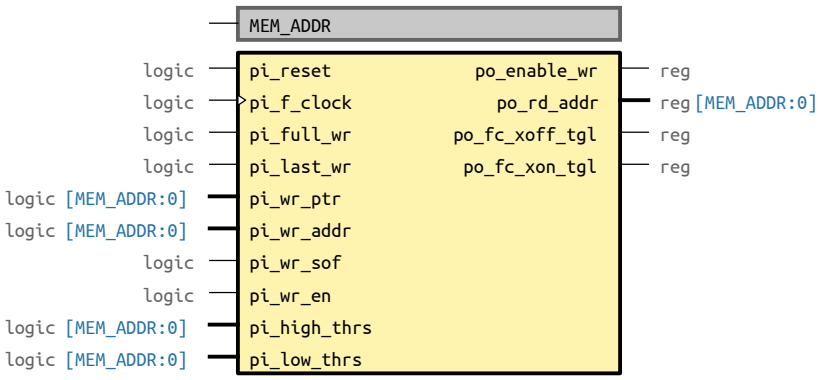


Fig. 37: Block Diagram of ip_mac_rx_fifo_g

Overview

The EMAC Receive FIFO Control module is responsible to generate all the control signals necessary to transfer the data form the EMAC Receive Memory module to the EMAC Asynchronous FIFO module. The EMAC Receive FIFO Control module provides the memory write memory pointer and the write address, used by the EMAC Receive module in order to calculate if the memory is ready (actual frame transmission begins after the internal Receive memory had reached either a programmable threshold or after a full frame is contained in the memory). The write address is updated (takes the write pointer value) whenever the memory contains enough data for Receive. The pointer update is made when the number of words of the frame exceeds a programmed threshold value or the entire frame is in the memory.

Table 50: Parameters

Name	Default value	Description
MEM_ADDR	6	Transmit memory address width (9 -> 512 locations, 10->1024)

Table 51: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)
pi_f_clock	input	wire logic	Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_full_wr	input	wire logic	Signals from/to Asynchronous FIFO Asynchronous FIFO full (no write can be performed)
pi_last_wr	input	wire logic	Asynchronous FIFO last valid location
po_enable_wr	output	var reg	Asynchronous FIFO write enable
pi_wr_ptr	input	wire logic[MEM_-ADDR:0]	Signals from MAC State Machine Used by RX FIFO to compute the memory state (full/empty/ready)
pi_wr_addr	input	wire logic[MEM_-ADDR:0]	Write address (memory write address)
po_rd_addr	output	var reg[MEM_-ADDR:0]	Used by EMAC Receive State to see the momory state (full/empty/ready)

continues on next page

Table 51 – continued from previous page

Name	Direction	Type	Description
pi_wr_sof	input	wire logic	Memory SOF and write enable (use for fc toggle function) Start of frame indication (resend a new FC packet since
pi_wr_en	input	wire logic	a new frame was received during pause period, a previously FC was send) FIFO write enable (valid start of frame)
pi_high_thrs	input	wire logic[MEM_-ADDR:0]	Flow control From configuration FC high threshold
pi_low_thrs	input	wire logic[MEM_-ADDR:0]	From configuration FC low threshold
po_fc_xoff_tgl	output	var reg	(to TX EMAC) insert XOFF flow control information
po_fc_xon_tgl	output	var reg	(to TX EMAC) insert XON flow control information

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Low/High Threshold Assignment

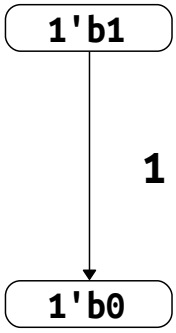


Table 52: FSM Transitions for high_low_en

#	Current State	Next State	Condition
1	1'b1	1'b0	[(!(~ pi_reset) && !(pi_high_thrs < fifo_level && pi_-wr_en == 1'b1 && pi_wr_sof == 1'b1))]

- always @(posedge pi_f_clock or negedge pi_reset)

Write Enable & Address/Pointer Update Process

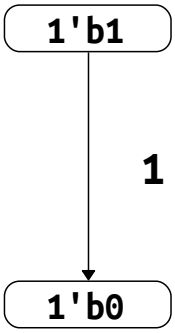


Table 53: FSM Transitions for po_enable_wr

#	Current State	Next State	Condition
1	1'b1	1'b0	[(!(~ pi_reset) && !(valid_data == 1'b0) && !(pi_full_wr == 1'b1))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_fifo : ip_mac_rx_fifo_g#(.MEM_ADDR(10))`

6.28 Module ip_mac_rx_gmii_g

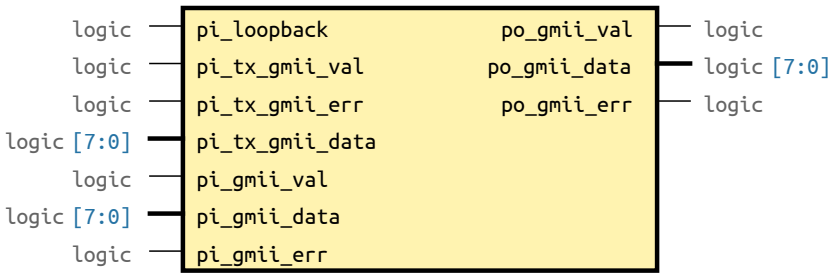


Fig. 38: Block Diagram of ip_mac_rx_gmii_g

Overview

The EMAC Receive MII/GMII module is responsible to multiplex the MII input interface (nibble oriented) signals coming from physical layer into GMII format when 10/100 Mbps operating mode is selected or to pass the GMII signal to the EMAC Receive State Machine when operating speed above 100 Mbps.

The EMAC Receive MII/GMII module it also checks for invalid Ethernet MAC frames by checking for proper byte-boundary alignment of the end of the frame. The block is responsible to generate the alignment error indication when frame length is not an integer number of bytes.

The module is also responsible for the internal loop-back operation, when the EMAC operates in loop-back mode. The EMAC Clock Manager module is responsible to switch between the input receive clock and transmit clock for loop-back operation.

Table 54: Ports

Name	Direction	Type	Description
pi_loopback	input	wire logic	Loopback mode select
pi_tx_gmii_val	input	wire logic	Loopback Info Loopback MII/GMII data valid indication (from TX)
pi_tx_gmii_err	input	wire logic	Loopback MII/GMII error indication (from TX)
pi_tx_gmii_data	input	wire logic[7:0]	Loopback MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from TX)
pi_gmii_val	input	wire logic	gmii Data Valid, Data Input Signals and Alignment Error Receive MII/GMII data valid indication (from PHY)
pi_gmii_data	input	wire logic[7:0]	Receive MII/GMII error indication (from PHY)
pi_gmii_err	input	wire logic	Receive MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from PHY)
po_gmii_val	output	wire logic	GMII Data Valid, Data Input Signals and Alignment Error Receive MII/GMII data valid indication (to RX state)
po_gmii_data	output	wire logic[7:0]	Receive MII/GMII error indication (to RX state)
po_gmii_err	output	wire logic	Receive MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (to RX state)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_gmii`

6.29 Module ip_mac_rx_hash_g

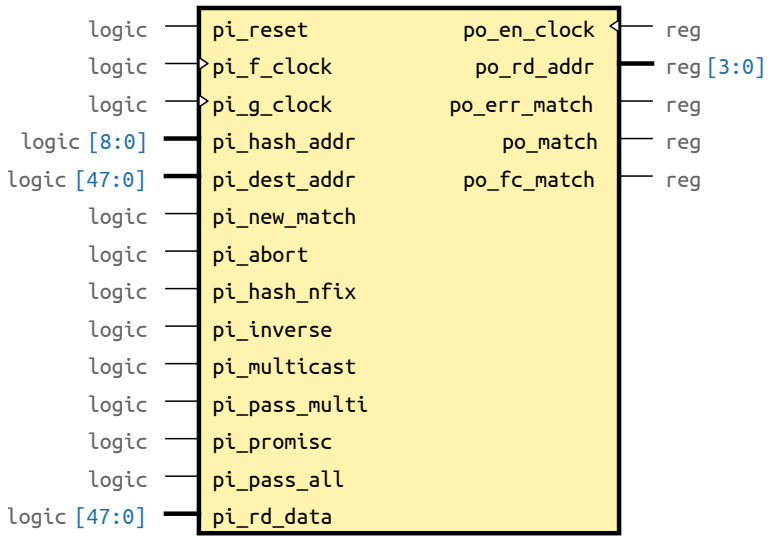


Fig. 39: Block Diagram of ip_mac_rx_hash_g

Overview

Imperfect filtering The EMAC Receive Hash/Exact Mach module is responsible for received frame filtering. For any incoming multicast frame, the EMAC applies the standard Ethernet cyclic redundancy check (CRC) function to the first 6 bytes that contain the destination address, then, if the hash filtering type is selected, the EMAC Receive Hash/Exact Mach module uses the most significant 9 bits (for a 512 bit hash table) of the result as a bit index into a table. If the indexed bit is set, the multicast frame is accepted. If the bit is cleared, the multicast frame is rejected. This filtering mode is called imperfect because frames not addressed to this station may slip through, but it still decreases the number of frames that the host can receive.

Perfect filtering The EMAC interprets a setup frame buffer in perfect filtering mode if the configuration bits are set accordingly. The EMAC can store 16 (programmable) destination addresses (full 48-bit Ethernet addresses). The EMAC compares the addresses of any incoming frame to these addresses, and also tests the status of the inverse filtering. It rejects addresses that: - Do not match if inverse filtering - Match if perfect filtering is set The setup frame must supply all 16 (programmable) addresses. Any mix of physical and multicast addresses can be used. Unused addresses should duplicate one of the valid addresses.

Table 55: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Software/Hardware Reset (receive clock domain)
pi_f_clock	input	wire logic	Free Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_g_clock	input	wire logic	Gated Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_en_clock	output	var reg	Receive GMII/MII 125/25/2.5 MHz clock gated clock enable
pi_hash_addr	input	wire logic[8:0]	From Receive EMAC Hash table index (9-MSB of CRC calculation over 48-bit destination address)

continues on next page

Table 55 – continued from previous page

Name	Direction	Type	Description
pi_dest_addr	input	wire logic[47:0]	Exact match (destination address of the frame)
pi_new_match	input	wire logic	Start a new search (match address, or hash filtering, toggle signal when new match should be prformed)
pi_abort	input	wire logic	Abort search (due to errors)
pi_hash_nfix	input	wire logic	Configuration Hash filtering + 1 Address match / 16 Address match
pi_inverse	input	wire logic	Inverse match
pi_multicast	input	wire logic	When asserted the imperfect filtering refers only for multicast addressees
pi_pass_multi	input	wire logic	Pass all multicast addresses
pi_promisc	input	wire logic	Promiscuos Mode (no DA filter)
pi_pass_all	input	wire logic	Pass all frames (bad or good)
pi_rd_data	input	wire logic[47:0]	Hash Table Memory access Memory (hash table) read data
po_rd_addr	output	var reg[3:0]	Memory (hash table) read address
po_err_match	output	var reg	Match Address Hit Address match output error
po_match	output	var reg	Address match output
po_fc_match	output	var reg	Address match output (control frame address)

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Gated Clock Enable

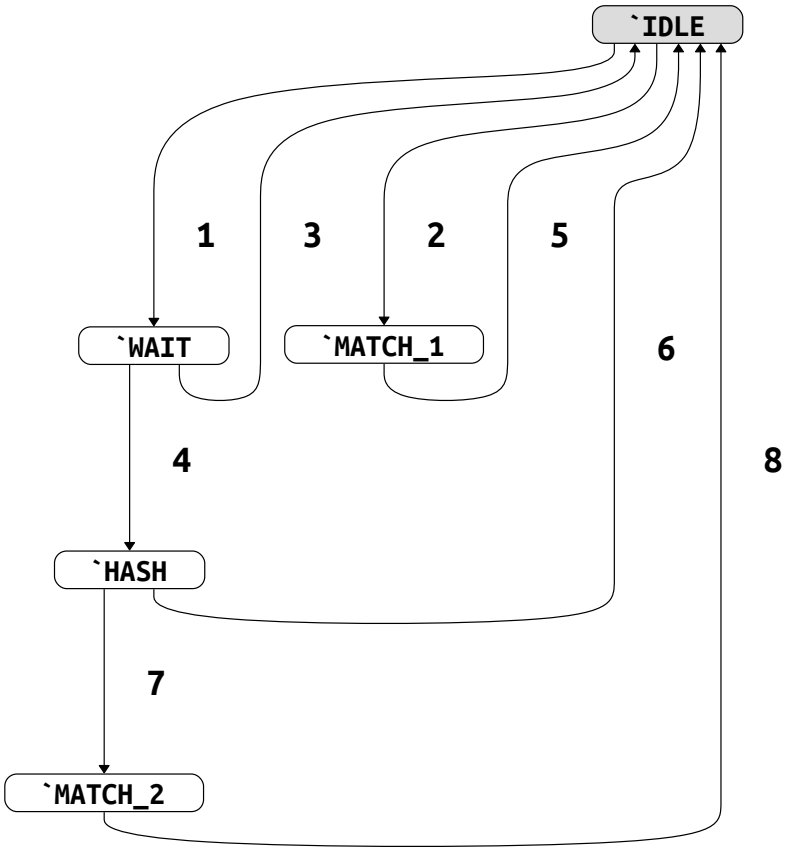


Table 56: FSM Transitions for fsm_hash_st

#	Current State	Next State	Condition
1	`IDLE	`WAIT	[(!(~ pi_reset) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_dest_addr == 48'h0180C2000001) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_promisc == 1'b1 || pi_pass_all == 1'b1) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_pass_multi == 1'b1 && pi_dest_addr[40] == 1'b1) && (pi_abort == 1'b0 && new_match == 1'b1 && pi_hash_nfix == 1'b1))]
2	`IDLE	`MATCH_1	[(!(~ pi_reset) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_dest_addr == 48'h0180C2000001) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_promisc == 1'b1 || pi_pass_all == 1'b1) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_pass_multi == 1'b1 && pi_dest_addr[40] == 1'b1) && !(pi_abort == 1'b0 && new_match == 1'b1 && pi_hash_nfix == 1'b1) && (pi_abort == 1'b0 && new_match == 1'b1))]
3	`WAIT	`IDLE	[(!(~ pi_reset) && (pi_abort == 1'b1))]
4	`WAIT	`HASH	[(!(~ pi_reset) && !(pi_abort == 1'b1))]
5	`MATCH_1	`IDLE	[(!(~ pi_reset) && (pi_abort == 1'b1)), (!(~ pi_reset) && !(pi_abort == 1'b1) && (pi_rd_data == pi_dest_addr)), (!(~ pi_reset) && !(pi_abort == 1'b1) && !(pi_rd_data == pi_dest_addr) && (po_rd_addr == 4'h0))]

continues on next page

Table 56 – continued from previous page

#	Current State	Next State	Condition
6	`HASH	`IDLE	[(!(~ pi_reset) && (pi_abort == 1'b1)), (!(~ pi_reset) && !(pi_abort == 1'b1) && (pi_rd_data[hash_idx] == 1'b1 && pi_dest_addr[40] == 1'b1 ‖ pi_multicast == 1'b0))]
7	`HASH	`MATCH_2	[(!(~ pi_reset) && !(pi_abort == 1'b1) && !(pi_rd_data[hash_idx] == 1'b1 && pi_dest_addr[40] == 1'b1 ‖ pi_multicast == 1'b0))]
8	`MATCH_2	`IDLE	[(!(~ pi_reset) && (pi_abort == 1'b1)), (!(~ pi_reset) && (po_rd_addr == 4'he) && !(pi_rd_data[47 : 32] == pi_dest_addr[47 : 32])), (!(~ pi_reset) && (po_rd_addr == 4'hf) && !(pi_rd_data[47 : 32] == pi_dest_addr[31 : 16])), (!(~ pi_reset) && (po_rd_addr == 4'h0) && (pi_rd_data[47 : 32] == pi_dest_addr[15 : 0])), (!(~ pi_reset) && (po_rd_addr == 4'h0) && !(pi_rd_data[47 : 32] == pi_dest_addr[15 : 0]))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_hash`

6.30 Module ip_mac_rx_state_g

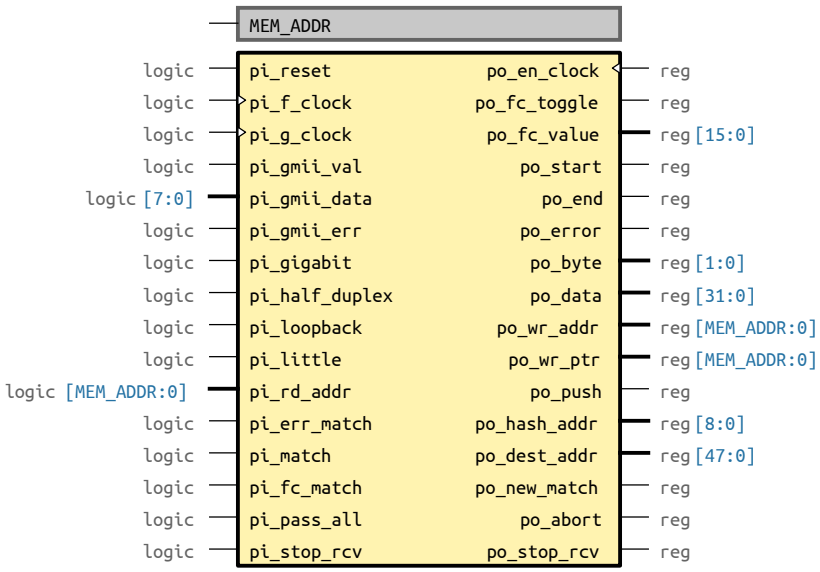


Fig. 40: Block Diagram of ip_mac_rx_state_g

Table 57: Parameters

Name	Default value	Description
MEM_ADDR	6	Receive memory address width (9 -> 512 locations, 10->1024)

Table 58: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)
pi_f_clock	input	wire logic	Free Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_g_clock	input	wire logic	Gated Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_en_clock	output	var reg	Enable Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_gmii_val	input	wire logic	gmii Data Valid, Data Input Signals and Alignment Error Receive MII/GMII data valid indication (from interface MII block)
pi_gmii_data	input	wire logic[7:0]	Receive MII/GMII error indication (from GMII block)
pi_gmii_err	input	wire logic	Receive MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from interface MII block)
pi_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_half_duplex	input	wire logic	Operating Half Duplex mode
pi_loopback	input	wire logic	Loopback information (frame received in loopback mode)
pi_little	input	wire logic	Little endian data format (used for statistic word translation)

continues on next page

Table 58 – continued from previous page

Name	Direction	Type	Description
po_fc_toggle	output	var reg	Flow control Interface New pause frame received
po_fc_value	output	var reg[15:0]	Pause time (from FC frame decoding FSM)
po_start	output	var reg	Data Path Interface Start of data frame indication
po_end	output	var reg	End of data frame indication
po_error	output	var reg	Error indication (valid when end of frame or indicate statistic word)
po_byte	output	var reg[1:0]	Byte enable command, valid only when end of frame
po_data	output	var reg[31:0]	FIFO Data bus (frame data 32-bit word)
pi_rd_addr	input	wire logic[MEM_-ADDR:0]	FIFO Control Interface Used by EMAC Receive State to see the momory state (full/empty/ready)
po_wr_addr	output	var reg[MEM_-ADDR:0]	Write address (memory write address)
po_wr_ptr	output	var reg[MEM_-ADDR:0]	Used by RX FIFO to compute the memory state (full/empty/ready)
po_push	output	var reg	Write enable command
pi_err_match	input	wire logic	Filterring Interface Address match error
pi_match	input	wire logic	Address match
pi_fc_match	input	wire logic	Address match (control frame address)
pi_pass_all	input	wire logic	Pass all frames (pass bad frames)
po_hash_addr	output	var reg[8:0]	Hash table index (9-MSB of CRC calculation over 48-bit destination address)
po_dest_addr	output	var reg[47:0]	Exact match (destination address of the frame)
po_new_match	output	var reg	Start a new search (match address, or hash filtering, toggle signal when new match should be prformed)
po_abort	output	var reg	Abort search (due to errors)
pi_stop_rcv	input	wire logic	Receive start/stop Receive MAC, receive stopped indication
po_stop_rcv	output	var reg	Receive MAC, receive stop command

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

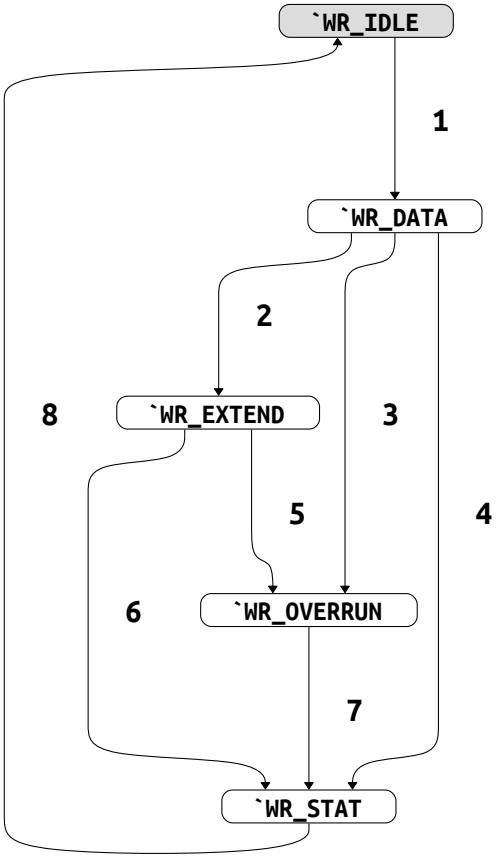


Table 59: FSM Transitions for wr_mac_state

#	Current State	Next State	Condition
1	`WR_IDLE	`WR_DATA	[(!(~ pi_reset) && (po_stop_rcv == 1'b0 && gmii_-valid == 1'b1 && gmii_eop == 1'b0))]
2	`WR_DATA	`WR_EXTEND	[(!(~ pi_reset) && (gigabit_hd == 1'b1 && extend_ok == 1'b0 && gmii_eop == 1'b1))]
3	`WR_DATA	`WR_OVERRUN	[(!(~ pi_reset) && !(gigabit_hd == 1'b1 && extend_-ok == 1'b0 && gmii_eop == 1'b1) && (gmii_eop == 1'b1) && (full == 1'b1)), (!(~ pi_reset) && !(gigabit_-hd == 1'b1 && extend_ok == 1'b0 && gmii_eop == 1'b1) && !(gmii_eop == 1'b1) && (counter[1 : 0] == 2'b00 && full == 1'b1))]
4	`WR_DATA	`WR_STAT	[(!(~ pi_reset) && !(gigabit_hd == 1'b1 && extend_-ok == 1'b0 && gmii_eop == 1'b1) && (gmii_eop == 1'b1) && !(full == 1'b1))]
5	`WR_EXTEND	`WR_OVERRUN	[(!(~ pi_reset) && (extend_ok == 1'b1) && (full == 1'b1)), (!(~ pi_reset) && !(extend_ok == 1'b1) && (gmii_ext == 1'b0) && (full == 1'b1))]
6	`WR_EXTEND	`WR_STAT	[(!(~ pi_reset) && (extend_ok == 1'b1) && !(full == 1'b1)), (!(~ pi_reset) && !(extend_ok == 1'b1) && (gmii_ext == 1'b0) && !(full == 1'b1))]
7	`WR_OVERRUN	`WR_STAT	[(!(~ pi_reset) && (full == 1'b0 && gmii_valid == 1'b0 && int_val == 1'b0))]
8	`WR_STAT	`WR_IDLE	[!(~ pi_reset)]

Functions

- crc32_data8 (logic[31:0] crc, logic[7:0] data)

CRC: 32 DATA: 8, POLY: 104C11DB7 next crc

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_state : ip_mac_rx_state_g#(.MEM_ADDR(10))`

6.31 Module ip_mac_rx_sync_g

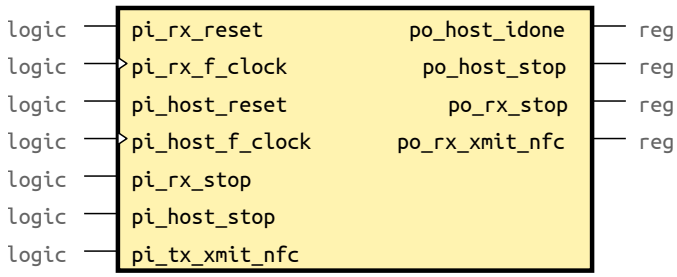


Fig. 41: Block Diagram of ip_mac_rx_sync_g

Table 60: Ports

Name	Direction	Type	Description
pi_rx_reset	input	wire logic	Receive clock and reset Global Hardware/-Software reset (receive clock domain, active low)
pi_rx_f_clock	input	wire logic	Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_host_idone	output	var reg	Reset done Receive initialisation done (host clock domain)
pi_host_reset	input	wire logic	Host clock and reset Global Hardware/Software reset (host clock domain, active low)
pi_host_f_clock	input	wire logic	Host clock (from Clock Manager)
pi_rx_stop	input	wire logic	Inputs -> Synchronized outputs Receive stop command acknowledge (receive clock domain)
po_host_stop	output	var reg	Receive stop command acknowledge (synchronized to host clock domain)
pi_host_stop	input	wire logic	Receive stop command (host clock domain)
po_rx_stop	output	var reg	Receive stop command (synchronized to receive clock domain)
pi_tx_xmit_nfc	input	wire logic	Transmit FSM data frame transmit enable (transmit clock domain)
po_rx_xmit_nfc	output	var reg	NOTE: This signal is not asserted during flow control frame transmission Transmit FSM data frame transmit enable (synchronized to receive clock domain)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > `rx_sync`

6.32 Module ip_mac_rx_top_g

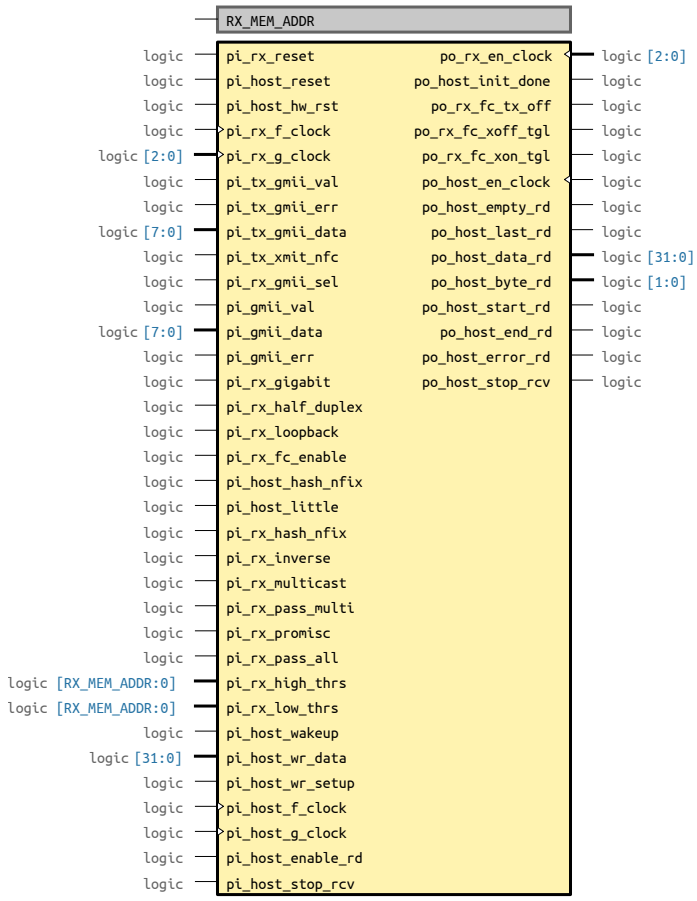


Fig. 42: Block Diagram of ip_mac_rx_top_g

Table 61: Parameters

Name	Default value	Description
RX_MEM_ADDR	6	Receive memory address width (9 -> 512 locations, 10->1024)

Table 62: Ports

Name	Direction	Type	Description
pi_rx_reset	input	wire logic	Global Software/Hardware Reset (receive clock domain)
pi_host_reset	input	wire logic	Global Software/Hardware Reset (host clock domain)
pi_host_hw_rst	input	wire logic	Global Hardware Reset (host clock domain)
pi_rx_f_clock	input	wire logic	Receive clock Free Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_rx_g_clock	input	wire logic[2:0]	Gated Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_rx_en_clock	output	wire logic[2:0]	Enable Receive GMII/MII 125/25/2.5 MHz clock (from Clock Manager)

continues on next page

Table 62 – continued from previous page

Name	Direction	Type	Description
po_host_init_done	output	wire logic	Initialisation done Receive initialisation done (host clock domain)
pi_tx_gmii_val	input	wire logic	Loopback GMII/MII Data Valid, Data Input Signals and Alignment Error Loopback MII/GMII data valid indication (from TX)
pi_tx_gmii_err	input	wire logic	Loopback MII/GMII error indication (from TX)
pi_tx_gmii_data	input	wire logic[7:0]	Loopback MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from TX)
pi_tx_xmit_nfc	input	wire logic	Transmit full duplex data frame transmit enable pending (non flow control frame is transmitted) Transmit FSM data frame transmit enable (transmit clock domain)
pi_rx_gmii_sel	input	wire logic	NOTE: This signal is not asserted during flow control frame transmission GMII/MII Data Valid, Data Input Signals and Alignment Error Receive GMII data select (demultiplex MII interface indication)
pi_gmii_val	input	wire logic	Note: po_rx_gmii_sel is balanced with the internal receive clock Receive MII/GMII data valid indication (from PHY)
pi_gmii_data	input	wire logic[7:0]	Receive MII/GMII error indication (from PHY)
pi_gmii_err	input	wire logic	Receive MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from PHY)
pi_rx_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_rx_half_duplex	input	wire logic	Operating Half Duplex mode
pi_rx_loopback	input	wire logic	Loopback mode select
pi_rx_fc_enable	input	wire logic	Receive flow control enable (flow control decoding enable)
po_rx_fc_tx_off	output	wire logic	Received FC packet (Transmit stop command)
po_rx_fc_xoff_tgl	output	wire logic	(to TX EMAC) insert XOFF flow control information
po_rx_fc_xon_tgl	output	wire logic	(to TX EMAC) insert XON flow control information
pi_host_hash_nfix	input	wire logic	Hash filtering + 1 Address match / 16 Address match
pi_host_little	input	wire logic	Little endian (data path organisation)
pi_rx_hash_nfix	input	wire logic	Configuration Filtering Hash/Exact Hash filtering + 1 Address match / 16 Address match
pi_rx_inverse	input	wire logic	Inverse match filtering mode
pi_rx_multicast	input	wire logic	When asserted the imperfect filtering refers only for multicast addressees
pi_rx_pass_multi	input	wire logic	Pass all multicast addresses
pi_rx_promisc	input	wire logic	Promiscuos Mode (no DA filter)
pi_rx_pass_all	input	wire logic	pass all bad frames (including FC frames)
pi_rx_high_thrs	input	wire logic[RX_-MEM_ADDR:0]	from configuration FC high threshold
pi_rx_low_thrs	input	wire logic[RX_-MEM_ADDR:0]	from configuration FC low threshold

continues on next page

Table 62 – continued from previous page

Name	Direction	Type	Description
pi_host_wakeup	input	wire logic	From/To Receive DMA (setup frame) Wake-up internal clock used by the Setup Frame FSM,
pi_host_wr_data	input	wire logic[31:0]	should be asserted at least 2 clock cycles (HOST clock) before asserting the pi_-host_wr_setup (write enable) and can be deasserted 2 clock cycles after Setup Frame completition Setup Frame data (HOST clock synchronous)
pi_host_wr_setup	input	wire logic	Setup Frame write enable (HOST clock synchronous)
pi_host_f_clock	input	wire logic	Receive data path HOST interface Free HOST interface clock signal
pi_host_g_clock	input	wire logic	Gated Free HOST interface clock signal
po_host_en_clock	output	wire logic	Enable Free HOST interface clock signal
pi_host_enable_rd	input	wire logic	Receive MAC data path, HOST enable command
po_host_empty_rd	output	wire logic	Receive MAC data path, HOST FIFO empty indication
po_host_last_rd	output	wire logic	Receive MAC data path, HOST FIFO last location indication
po_host_data_rd	output	wire logic[31:0]	Receive MAC data path, HOST data (transmit data)
po_host_byte_rd	output	wire logic[1:0]	Receive MAC data path, HOST byte enable (transmit data byte enable)
po_host_start_rd	output	wire logic	Receive MAC data path, HOST start of frame indication
po_host_end_rd	output	wire logic	Receive MAC data path, HOST start of frame indication
po_host_error_rd	output	wire logic	Receive MAC data path, HOST frame error indication (asserted when
pi_host_stop_rcv	input	wire logic	frame is bigger than 64 bytes and receive MAC cannot drop the frame or pass bad frames mode is selected by asserting pi_-emac_pass_all) Receive Start/Stop Receive MAC, receive stopped indication (HOST clock synchronous)
po_host_stop_rcv	output	wire logic	Receive MAC, receive stop command (HOST clock synchronous)

Instances

- *ip_emac_top* > *ip_mac_top_g* > `mac_rx_top : ip_mac_rx_top_g#(.RX_MEM_ADDR(10))`

Submodules

- **ip_mac_rx_top_g#(.RX_MEM_ADDR(10))**
  - cfg_hash : *ip_mac_cfg_hash_g*
  - fc_dec : *ip_mac_fc_dec_g*
  - rx_async : *ip_async_fifo_g* #(.MEM_WIDTH(37))
  - rx_data_dram : *ip_mac_dram_002* #(.MEM_ADDR(10), .MEM_WIDTH(37))
  - rx_dram_hash_0 : *ip_mac_dram_004* #(.MEM_ADDR(4), .MEM_WIDTH(32))
  - rx_dram_hash_1 : *ip_mac_dram_003* #(.MEM_ADDR(4), .MEM_WIDTH(16))
  - rx_endian : *ip_mac_big_endian*
  - rx_fifo : *ip_mac_rx_fifo_g* #(.MEM_ADDR(10))
  - rx_gmii : *ip_mac_rx_gmii_g*

- rx_hash : *ip_mac_rx_hash_g*
- rx_state : *ip_mac_rx_state_g* #(.MEM_ADDR(10))
- rx_sync : *ip_mac_rx_sync_g*

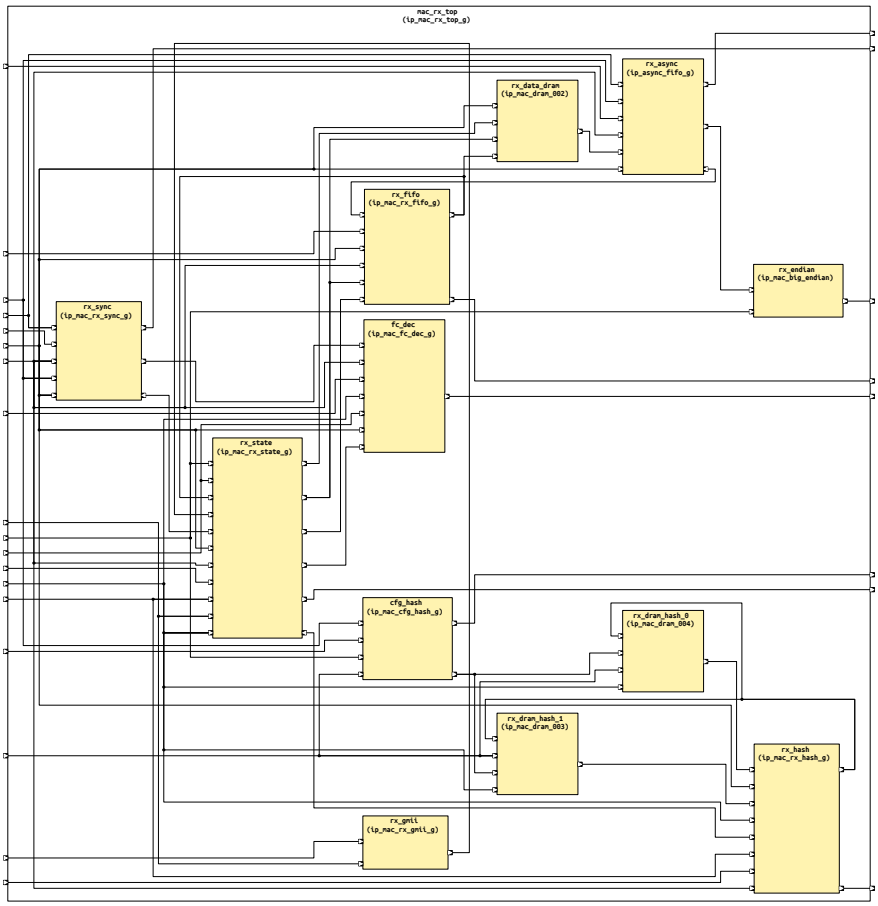


Fig. 43: Flow Diagram of ip_mac_rx_top_g

6.33 Module ip_mac_top_g

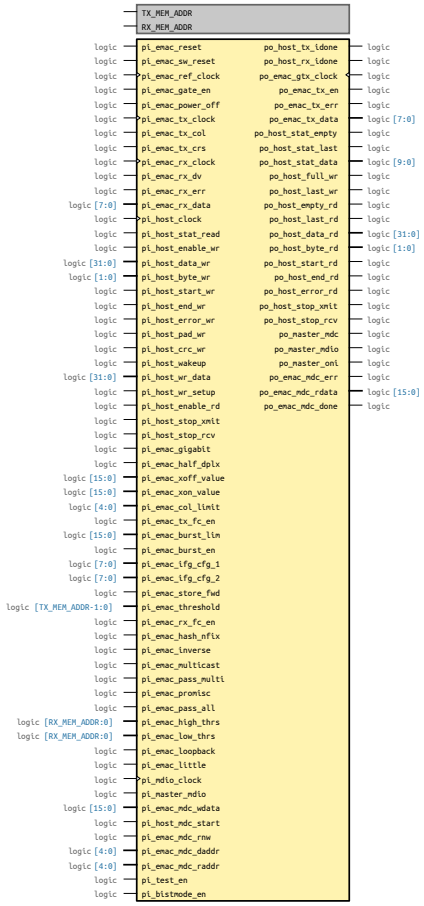


Fig. 44: Block Diagram of ip_mac_top_g

Table 63: Parameters

Name	Default value	Description
TX_MEM_ADDR	9	Transmit memory address width (9 -> 512 locations, 10->1024)
RX_MEM_ADDR	6	Receive memory address width (9 -> 512 locations, 10->1024)

Table 64: Ports

Name	Direction	Type	Description
pi_emac_reset	input	wire logic	Global Hardware reset (active low)
pi_emac_sw_reset	input	wire logic	Global Software reset (active high) (should be applied whenever
pi_emac_ref_clock	input	wire logic	GMII 125 MHz reference clock
pi_emac_gate_en	input	wire logic	Auto Gating Clock Enable (power saving)
pi_emac_power_-off	input	wire logic	Power off (all internal clocks are disabled, except for the HOST clock)
po_host_tx_idone	output	wire logic	Transmit/Receive reset (initialization) done Transmit initialization done (host clock domain)

continues on next page

Table 64 – continued from previous page

Name	Direction	Type	Description
po_host_rx_idone	output	wire logic	Receive initialization done (host clock domain)
pi_emac_tx_clock	input	wire logic	Transmit GMII/MII interface Transmit MII 25/2.5 MHz clock (from PHY)
po_emac_gtx_-clock	output	wire logic	Transmit GMII 125 MHz clock (to PHY)
po_emac_tx_en	output	wire logic	Transmit MII/GMII enable indication (to PHY)
po_emac_tx_err	output	wire logic	Transmit MII/GMII error indication (to PHY)
po_emac_tx_data	output	wire logic[7:0]	Transmit MII/GMII data (MII data is po_-emac_tx_data[3:0]) (to PHY)
pi_emac_tx_col	input	wire logic	Collision indication (from PHY)
pi_emac_tx_crs	input	wire logic	Carrier Sense indication (from PHY)
pi_emac_rx_clock	input	wire logic	Receive GMII/MII interface Receive GMI-I/MII 125/25/2.5 MHz clock (from PHY)
pi_emac_rx_dv	input	wire logic	Receive MII/GMII data valid indication (from PHY)
pi_emac_rx_err	input	wire logic	Receive MII/GMII error indication (from PHY)
pi_emac_rx_data	input	wire logic[7:0]	Receive MII/GMII data (MII data is pi_-emac_rx_data[3:0]) (from PHY)
pi_host_clock	input	wire logic	HOST interface (common) HOST interface clock signal
pi_host_stat_read	input	wire logic	Transmit statistic HOST interface Statistic TDES0 word [8] = jabber,[7] = late collision, [6] = excess collision,[5:2] = collision counter [1] = underrun,[0] = deferred Read statistic word command
po_host_stat_-empty	output	wire logic	Statistic FIFO not empty
po_host_stat_last	output	wire logic	Last statistic word indication
po_host_stat_data	output	wire logic[9:0]	Statistic TDES0 word data
pi_host_enable_wr	input	wire logic	Tramsmit data path HOST interface Transmit MAC data path, HOST enable command
po_host_full_wr	output	wire logic	Transmit MAC data path, HOST FIFO full indication
po_host_last_wr	output	wire logic	Transmit MAC data path, HOST FIFO last location indication
pi_host_data_wr	input	wire logic[31:0]	Transmit MAC data path, HOST data (transmit data)
pi_host_byte_wr	input	wire logic[1:0]	Transmit MAC data path, HOST byte enable (transmit data byte enable)
pi_host_start_wr	input	wire logic	Transmit MAC data path, HOST start of frame indication
pi_host_end_wr	input	wire logic	Transmit MAC data path, HOST start of frame indication
pi_host_error_wr	input	wire logic	Unused please check if useful (if not remove)
pi_host_pad_wr	input	wire logic	Transmit MAC data path, HOST padding enable (valid only when pi_host_end_wr)
pi_host_crc_wr	input	wire logic	Transmit MAC data path, HOST crc enable (valid only when pi_host_end_wr)

continues on next page

Table 64 – continued from previous page

Name	Direction	Type	Description
pi_host_wakeup	input	wire logic	Receive setup frame interface (not affected by software reset, setup frame is written during software reset) Wake-up internal clock used by the Setup Frame FSM,
pi_host_wr_data	input	wire logic[31:0]	should be asserted at least 2 clock cycles (HOST clock) before asserting the pi_-host_wr_setup (write enable) and can be deasserted 2 clock cycles after Setup Frame completition Setup Frame data (HOST clock synchronous)
pi_host_wr_setup	input	wire logic	Setup Frame write enable (HOST clock synchronous)
pi_host_enable_rd	input	wire logic	Receive data path HOST interface Receive MAC data path, HOST enable command
po_host_empty_rd	output	wire logic	Receive MAC data path, HOST FIFO empty indication
po_host_last_rd	output	wire logic	Receive MAC data path, HOST FIFO last location indication
po_host_data_rd	output	wire logic[31:0]	Receive MAC data path, HOST data (transmit data)
po_host_byte_rd	output	wire logic[1:0]	Receive MAC data path, HOST byte enable (transmit data byte enable)
po_host_start_rd	output	wire logic	Receive MAC data path, HOST start of frame indication
po_host_end_rd	output	wire logic	Receive MAC data path, HOST start of frame indication
po_host_error_rd	output	wire logic	Receive MAC data path, HOST frame error indication (asserted when
po_host_stop_xmit	output	wire logic	frame is bigger than 64 bytes and receive MAC cannot drop the frame or pass bad frames mode is selected by asserting pi_emac_pass_all) Start/Stop transmit and receive process Transmit MAC, transmit stopped indication (HOST clock synchronous)
pi_host_stop_xmit	input	wire logic	Transmit MAC, transmit stop command (HOST clock synchronous)
po_host_stop_rcv	output	wire logic	Receive MAC, receive stopped indication (HOST clock synchronous)
pi_host_stop_rcv	input	wire logic	Receive MAC, receive stop command (HOST clock synchronous)
pi_emac_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_emac_half_dplx	input	wire logic	Operating Half Duplex mode
pi_emac_xoff_-value	input	wire logic[15:0]	Transmit configuration inputs (require software reset to be active during change) XOFF flow control pause value
pi_emac_xon_-value	input	wire logic[15:0]	XON flow control pause value
pi_emac_col_limit	input	wire logic[4:0]	Half Duplex back pressure collision limit
pi_emac_tx_fc_en	input	wire logic	(maximum collision number during back pressure algorithm) Transmit flow control enable
pi_emac_burst_lim	input	wire logic[15:0]	Burst limit (valid only when operating mode is 1000 Mbps)

continues on next page

Table 64 – continued from previous page

Name	Direction	Type	Description
pi_emac_burst_en	input	wire logic	Burst enable (valid only when operating mode is 1000 Mbps)
pi_emac_ifg_cfg_1	input	wire logic[7:0]	Interframe gap part 1 (usually 2/3 from IFG)
pi_emac_ifg_cfg_2	input	wire logic[7:0]	Interframe gap part 2 (usually 1/3 from IFG)
pi_emac_store_fwd	input	wire logic	Store and Forward transmit FIFO operating mode
pi_emac_threshold	input	wire logic[TX_-MEM_ADDR-1:0]	Cut Trough (pi_emac_store_fwd not asserted) FIFO threshold
pi_emac_rx_fc_en	input	wire logic	Receive configuration inputs (require software reset to be active during change) Receive flow control enable (flow control decoding enable)
pi_emac_hash_nfix	input	wire logic	Hash filtering + 1 Address match / 16 Address match
pi_emac_inverse	input	wire logic	Inverse match filtering mode
pi_emac_multicast	input	wire logic	When asserted the imperfect filtering refers only for multicast addressees
pi_emac_pass_-multi	input	wire logic	Pass all multicast addresses
pi_emac_promisc	input	wire logic	Promiscuous Mode (no DA filter)
pi_emac_pass_all	input	wire logic	pass all bad frames (including FC frames)
pi_emac_high_thrs	input	wire logic[RX_-MEM_ADDR:0]	from configuration FC high threshold
pi_emac_low_thrs	input	wire logic[RX_-MEM_ADDR:0]	from configuration FC low threshold
pi_emac_loopback	input	wire logic	Miscelanous configuration inputs (require software reset to be active during change) Loopback mode select
pi_emac_little	input	wire logic	Little endian (data path organization)
pi_mdio_clock	input	wire logic	Management Data Input/Output interface (MDIO) Master MDIO reference clock
po_master_mdc	output	wire logic	Master MDIO output 2.5 MHz clock
pi_master_mdio	input	wire logic	Master MDIO data input line (tri-state buffer outside of the module)
po_master_mdio	output	wire logic	Master MDIO data output line (tri-state buffer outside of the module)
po_master_oni	output	wire logic	Master MDIO direction tri-state buffer control (1 - Output, 0 - Input)
po_emac_mdc_err	output	wire logic	Indicates that a read from MDIO interface is invalid and the
pi_emac_mdc_-wdata	input	wire logic[15:0]	operation should be retried. This is indicated during a read turn-around cycle when the MDIO slave does not drive the MDIO signal to the low state. MDIO transaction complete. (MDIO clock domain, remain asserted till pi_host_mdc_start is deasserted) MDIO write data (not synchronized, false path, needs to be stable durring pi_host_mdc_start)
po_emac_mdc_-rdata	output	wire logic[15:0]	MDIO read data (not synchronized, false path, needs to be stable durring pi_host_-mdc_start)

continues on next page

Table 64 – continued from previous page

Name	Direction	Type	Description
pi_host_mdc_start	input	wire logic	(MDIO clock domain, remain stable till pi_host_mdc_start is deasserted) Setting this bit initiates an MDIO read/write operation (asynchronous,
po_emac_mdc_-done	output	wire logic	internally synchronized to MDC clock). Remain asserted till the transaction complete. MDIO transaction complete.
pi_emac_mdc_rnw	input	wire logic	(MDIO clock domain, remain asserted till pi_host_mdc_start is deasserted) This bit indicates the direction of the MDIO operation type:
pi_emac_mdc_-daddr	input	wire logic[4:0]	0 - MDIO Write operation, 1 - MDIO read operation (not synchronized, false path, needs to be stable during pi_host_mdc_-start) This field is used to specify the device address to be accessed.
pi_emac_mdc_-raddr	input	wire logic[4:0]	(not synchronized, false path, needs to be stable during pi_host_mdc_start) This field is used to specify the register address to be accessed.
pi_test_en	input	wire logic	(not synchronized, false path, needs to be stable during pi_host_mdc_start) Test and Scan interface signals Test mode enable
pi_bistmode_en	input	wire logic	

Instances

- *ip_emac_top* > `mac_top : ip_mac_top_g#(.TX_MEM_ADDR(10), .RX_MEM_ADDR(10))`

Submodules

- **ip_mac_top_g#(.TX_MEM_ADDR(10), .RX_MEM_ADDR(10))**
  - mac_clk_mng : *ip_mac_clk_mng_g*
  - mac_mdio : *ip_mac_mdio_g*
  - mac_rx_top : *ip_mac_rx_top_g* #(.RX_MEM_ADDR(10))
  - mac_tx_top : *ip_mac_tx_top_g* #(.TX_MEM_ADDR(10))

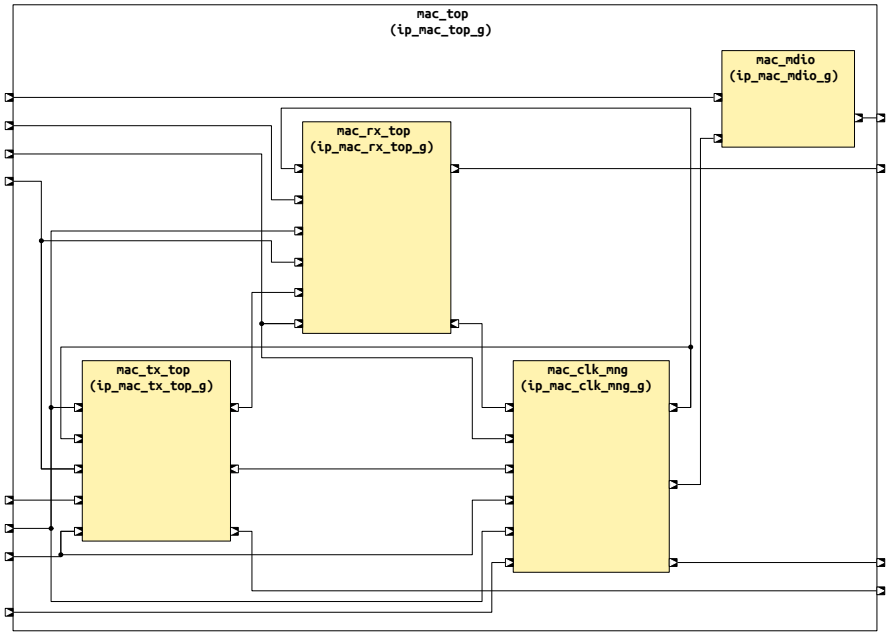


Fig. 45: Flow Diagram of ip_mac_top_g

6.34 Module ip_mac_tx_bkoff_g

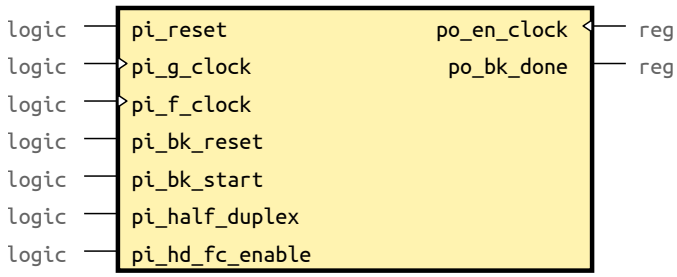


Fig. 46: Block Diagram of ip_mac_tx_bkoff_g

Overview

When a transmission attempt has terminated due to a collision, it is retried by the transmitting CSMA/CD sublayer until either it is successful or a maximum number of attempts have been made and all have terminated due to collisions. Note that all attempts to transmit a given frame are completed before any subsequent outgoing frames are transmitted. The scheduling of the retransmissions is determined by a controlled randomization process called truncated binary exponential backoff.

At the end of enforcing

a collision (jamming), the CSMA/CD sublayer delays before attempting to retransmit the frame. The delay is an integer multiple of 512 bit time slot. The number of slot times to delay before the nth retransmission attempt is chosen as a uniformly distributed random integer r in the range:

-1 < r < 2 power k, where k = min (n, 10)

If all n attempts limit fails, this event is reported as an error. Algorithms used to generate the integer r should be designed to minimize the correlation between the numbers generated by any two stations at any given time.

Note: The values given above define the most aggressive behavior that a station may exhibit in attempting to retransmit after a collision. In the course of implementing the retransmission scheduling procedure, a station may introduce extra delays that will degrade its own throughput, but in no case may a station retransmission scheduling result in a lower average delay between retransmission attempts than the procedure defined above.

Table 65: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)
pi_g_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, gated clock)
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, free clock)
po_en_clock	output	var reg	Transmit GMII/MII 125/25/2.5 MHz clock gated clock enable
pi_bk_reset	input	wire logic	Backoff interface Backoff counter reset command
pi_bk_start	input	wire logic	Backoff algorithm start command
pi_half_duplex	input	wire logic	Operating Half Duplex mode
pi_hd_fc_enable	input	wire logic	Half-Duplex Flow Control enable
po_bk_done	output	var reg	Backoff done indication

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Clock Gating Module

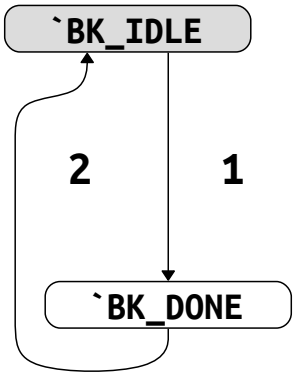


Table 66: FSM Transitions for bk_state

#	Current State	Next State	Condition
1	`BK_IDLE	`BK_DONE	[(!(~ pi_reset) && (pi_bk_start == 1'b1))]
2	`BK_DONE	`BK_IDLE	[(!(~ pi_reset) && (bk_ended == slot_cnt && pi_hd_-fc_enable == 1'b0)), (!(~ pi_reset) && !(bk_ended == slot_cnt && pi_hd_fc_enable == 1'b0) && (pi_hd_-fc_enable == 1'b1))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_bkoff`

6.35 Module ip_mac_tx_dpath_g

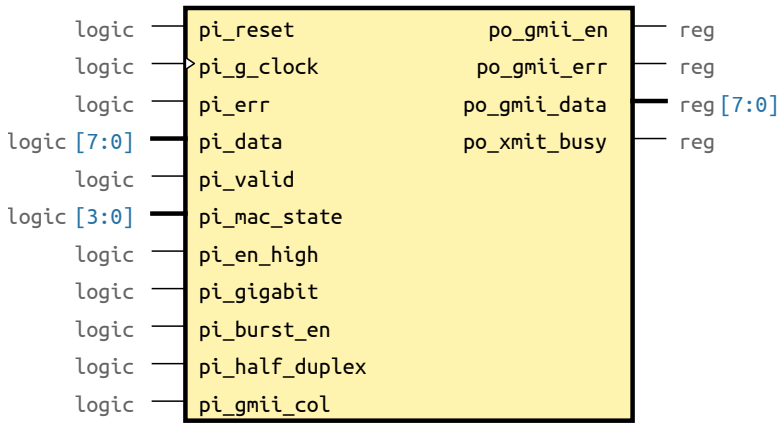


Fig. 47: Block Diagram of ip_mac_tx_dpath_g

Table 67: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Software/Hardware Reset (transmit clock domain)
pi_g_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_err	input	wire logic	Signals comming in from the data FIFO related to data transfer Error indication
pi_data	input	wire logic[7:0]	Data bus (frame data 8-bit word)
pi_valid	input	wire logic	Valid data
pi_mac_state	input	wire logic[3:0]	MAC state
pi_en_high	input	wire logic	Enable High (MSB)
pi_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_burst_en	input	wire logic	Burst enable (valid only when operating mode is 1000 Mbps)
pi_half_duplex	input	wire logic	GMII Interface Half duplex operating mode
pi_gmii_col	input	wire logic	Collision indication (from PHY)
po_gmii_en	output	var reg	Transmit MII/GMII enable indication (to PHY)
po_gmii_err	output	var reg	Transmit MII/GMII error indication (to PHY)
po_gmii_data	output	var reg[7:0]	Transmit MII/GMII data (MII data is po_emac_tx_data[3:0]) (to PHY)
po_xmit_busy	output	var reg	Used by the defering process (transmit enable ored with transmit error)

Functions

- crc32_data8 (logic[31:0] crc, logic[7:0] data)

CRC: 32 DATA: 8, POLY: 104C11DB7 next crc

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_dpath`

6.36 Module ip_mac_tx_dsplit_g

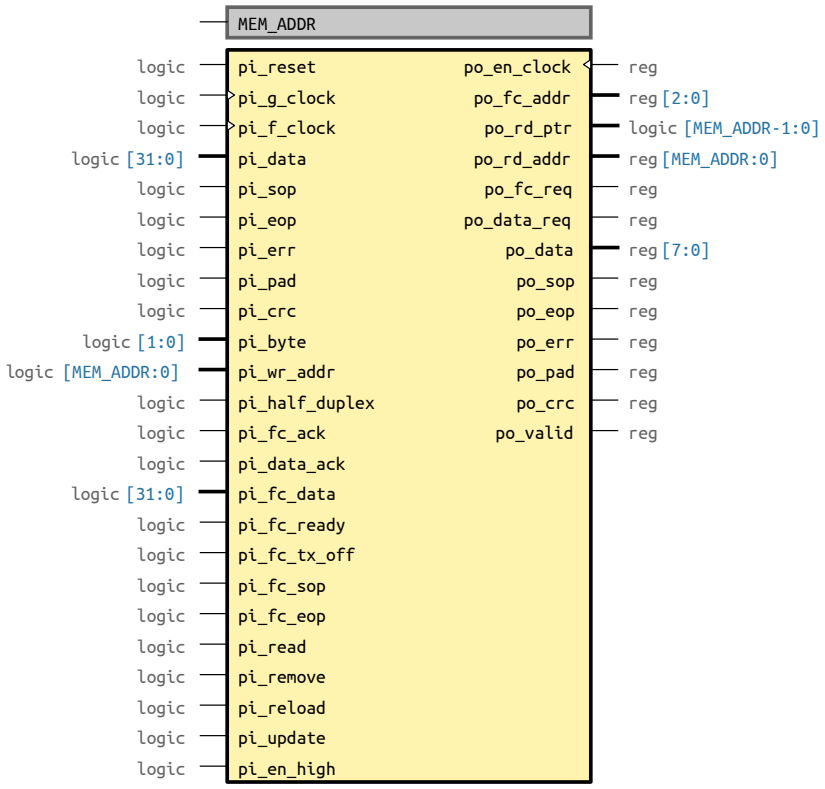


Fig. 48: Block Diagram of ip_mac_tx_dsplit_g

Table 68: Parameters

Name	Default value	Description
MEM_ADDR	6	Transmit memory address width (9 -> 512 locations, 10->1024)

Table 69: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Software/Hardware Reset (transmit clock domain)
pi_g_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, gated clock)
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, free clock)
po_en_clock	output	var reg	Transmit GMII/MII 125/25/2.5 MHz clock gated clock enable
pi_data	input	wire logic[31:0]	Signals comming in from the data FIFO related to data transfer FIFO Data bus (frame data 32-bit word)
pi_sop	input	wire logic	Start of data frame indication
pi_eop	input	wire logic	End of data frame indication
pi_err	input	wire logic	Error frame indication
pi_pad	input	wire logic	Pad append command, valid only when end of frame (When padding enable

continues on next page

Table 69 – continued from previous page

Name	Direction	Type	Description
pi_crc	input	wire logic	and frame has less than 64 bytes the CRC is appended regardless of the CRC append setting CRC append command, valid only when end of frame
pi_byte	input	wire logic[1:0]	Byte enable information
pi_wr_addr	input	wire logic[MEM_-ADDR:0]	Read & Write pointers TX FIFO write address information (used by tx state to determine the
po_fc_addr	output	var reg[2:0]	FIFO condition full/empty/ready) Flow control read address
pi_half_duplex	input	wire logic	Half duplex flow control enable Half duplex operating mode
po_rd_ptr	output	wire logic[MEM_-ADDR-1:0]	Output Signals to buffer manager related to data transfer Memory read address
po_rd_addr	output	var reg[MEM_-ADDR:0]	Used by TX FIFO to compute the memory state (full/empty/ready)
po_fc_req	output	var reg	Request & Acknowledge Flow control frame transmit request
po_data_req	output	var reg	Data frame transmit request
pi_fc_ack	input	wire logic	Flow control frame transmit acknowledge
pi_data_ack	input	wire logic	Data frame transmit acknowledge
pi_fc_data	input	wire logic[31:0]	Flow control generator interface Flow control frame data
pi_fc_ready	input	wire logic	New flow control frame ready
pi_fc_tx_off	input	wire logic	Flow control (pause frame was received, stop transmit command)
pi_fc_sop	input	wire logic	Flow control frame end
pi_fc_eop	input	wire logic	Flow control frame start
pi_read	input	wire logic	Read next data
pi_remove	input	wire logic	Remove current frame (drop frame)
pi_reload	input	wire logic	Reload current frame (retransmit)
pi_update	input	wire logic	Update read pointers (collision window out)
pi_en_high	input	wire logic	Enable High (MSB)
po_data	output	var reg[7:0]	Signals comming in from the data FIFO related to data transfer Output 8-bit data
po_sop	output	var reg	Start of data frame indication
po_eop	output	var reg	End of data frame indication
po_err	output	var reg	Error frame indication
po_pad	output	var reg	Pad append command, valid only when end of frame (When padding enable
po_crc	output	var reg	and frame has less than 64 bytes the CRC is appended regardless of the CRC append setting CRC append command, valid only when end of frame
po_valid	output	var reg	Valid output data

Always Blocks

- always @(posedge pi_g_clock or negedge pi_reset)

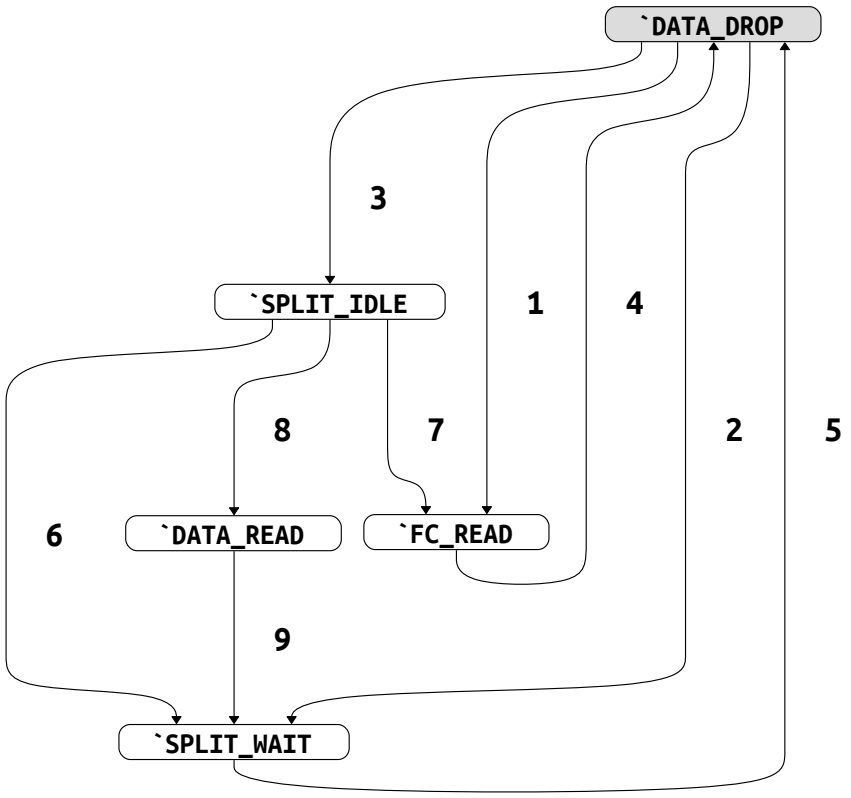


Table 70: FSM Transitions for split_state

#	Current State	Next State	Condition
1	`DATA_DROP	`FC_READ	[(!(~ pi_reset) && (pi_fc_ack == 1'b1))]
2	`DATA_DROP	`SPLIT_WAIT	[(!(~ pi_reset) && !(pi_fc_ack == 1'b1) && (reload_-frame == 1'b1)), (!(~ pi_reset) && !(pi_fc_ack == 1'b1) && !(reload_frame == 1'b1) && (valid_data == 1'b1 && pi_sop == 1'b0 || unlock_sop == 1'b0))]
3	`DATA_DROP	`SPLIT_IDLE	[(!(~ pi_reset) && !(pi_fc_ack == 1'b1) && !(reload_-frame == 1'b1) && !(valid_data == 1'b1 && pi_sop == 1'b0 || unlock_sop == 1'b0) && (valid_data == 1'b1 && pi_sop == 1'b1 && unlock_sop == 1'b1))]
4	`FC_READ	`DATA_DROP	[(!(~ pi_reset) && (remove_frame == 1'b1)), (!(~ pi_-reset) && !(remove_frame == 1'b1) && (reload_-frame == 1'b1)), (!(~ pi_reset) && !(remove_frame == 1'b1) && !(reload_frame == 1'b1) && (po_eop == 1'b1 && pi_read == 1'b1))]
5	`SPLIT_WAIT	`DATA_DROP	[(!(~ pi_reset) && !(reload_frame == 1'b1))]
6	`SPLIT_IDLE	`SPLIT_WAIT	[(!(~ pi_reset) && (reload_frame == 1'b1))]
7	`SPLIT_IDLE	`FC_READ	[(!(~ pi_reset) && !(reload_frame == 1'b1) && (pi_-fc_ack == 1'b1))]
8	`SPLIT_IDLE	`DATA_READ	[(!(~ pi_reset) && !(reload_frame == 1'b1) && !(pi_-fc_ack == 1'b1) && (pi_data_ack == 1'b1))]

continues on next page

Table 70 – continued from previous page

#	Current State	Next State	Condition
9	`DATA_READ	`SPLIT_WAIT	[(!(~ pi_reset) && (remove_frame == 1'b1)), (!(~ pi_reset) && !(remove_frame == 1'b1) && (reload_frame == 1'b1)), (!(~ pi_reset) && !(remove_frame == 1'b1) && !(reload_frame == 1'b1) && (po_eop == 1'b1 && valid_data == 1'b1 && pi_read == 1'b1)), (!(~ pi_reset) && !(remove_frame == 1'b1) && !(reload_frame == 1'b1) && !(po_eop == 1'b1 && valid_data == 1'b1 && pi_read == 1'b1) && !(split_cnt == 2'd0 && valid_data == 1'b1 && pi_read == 1'b1) && (po_eop == 1'b1 && valid_data == 1'b0 && pi_read == 1'b1))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_dsplit : ip_mac_tx_dsplit_g#(.MEM_ADDR(10))`

6.37 Module ip_mac_tx_fifo_g

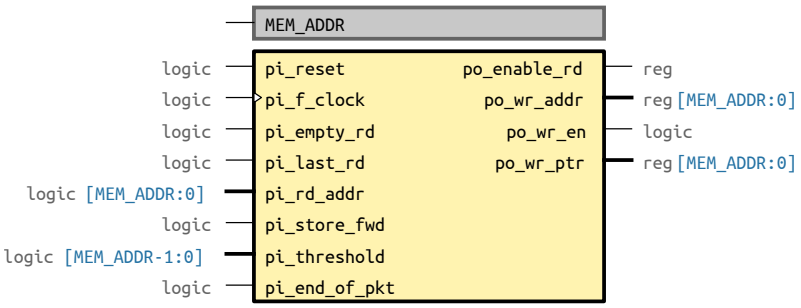


Fig. 49: Block Diagram of ip_mac_tx_fifo_g

Overview

The EMAC Transmit FIFO Control module is responsible to generate all the control signals necessary to transfer the data form the EMAC Asynchronous FIFO module to the EMAC Transmit Memory module. The EMAC Transmit FIFO Control module provides the memory write pointer and the write address, used by the EMAC Transmit module in order to calculate if the memory is ready (actual frame transmission begins after the internal transmit memory had reached either a programmable threshold or after a full frame is contained in the memory). The write address is updated (takes the write pointer value) whenever the memory contains enough data for transmit. The pointer update is made when the number of words of the frame exceeds a programmed threshold value or the entire frame is in the memory.

Table 71: Parameters

Name	Default value	Description
MEM_ADDR	6	Transmit memory address width (9 -> 512 locations, 10->1024)

Table 72: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, free clock)
pi_empty_rd	input	wire logic	Signals from/to Asynchronous FIFO Asynchronous FIFO empty (no read can be performed)
pi_last_rd	input	wire logic	Asynchronous FIFO last valid location
po_enable_rd	output	var reg	Asynchronous FIFO read enable
pi_rd_addr	input	wire logic[MEM_ADDR:0]	Signals from MAC State Machine Used by TX FIFO to compute the memory state (full/empty/ready)
po_wr_addr	output	var reg[MEM_ADDR:0]	Used by EMAC Transmit State to compute the memory state (full/empty/ready)
po_wr_en	output	wire logic	Signals to memory (synchronous FIFO block) Write memory enable
po_wr_ptr	output	var reg[MEM_ADDR:0]	Write pointer used by the memory to store data

continues on next page

Table 72 – continued from previous page

Name	Direction	Type	Description
pi_store_fwd	input	wire logic	Configuration Interface Store and forward / Cut trught
pi_threshold	input	wire logic[MEM_-ADDR-1:0]	Cut trught threshold
pi_end_of_pkt	input	wire logic	The end of packet/frame is stored into memory

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Assign Read async FIFO/Write sync FIFO enable

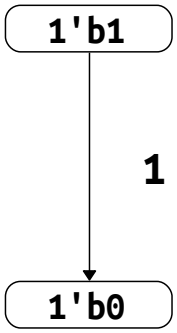


Table 73: FSM Transitions for po_enable_rd

#	Current State	Next State	Condition
1	1'b1	1'b0	[(!(~ pi_reset) && !(po_wr_ptr[MEM_ADDR] != pi_rd_addr[MEM_ADDR] && po_wr_ptr[MEM_-ADDR - 1 : 0] == pi_rd_addr[MEM_ADDR - 1 : 0]) && !(wr_inc[MEM_ADDR] != pi_rd_addr[MEM_-ADDR] && wr_inc[MEM_ADDR - 1 : 0] == pi_rd_-addr[MEM_ADDR - 1 : 0] && po_wr_en == 1'b1) && !(pi_empty_rd == 1'b1))]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_fifo : ip_mac_tx_fifo_g#(.MEM_-ADDR(10))`

6.38 Module ip_mac_tx_fsm_g

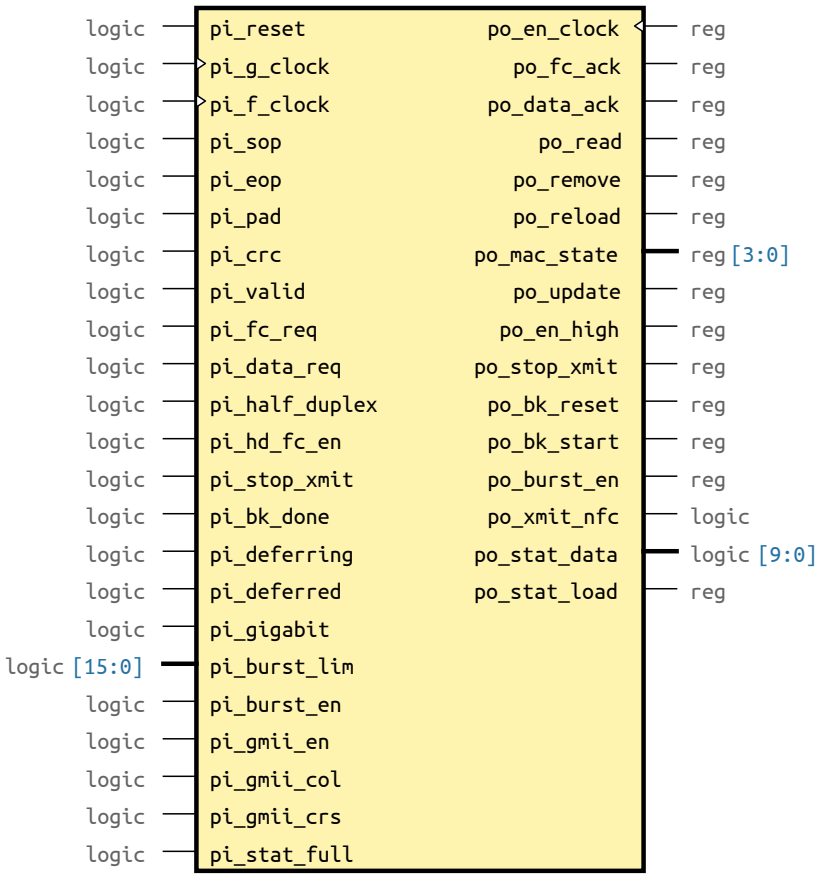


Fig. 50: Block Diagram of ip_mac_tx_fsm_g

Table 74: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Software/Hardware Reset (transmit clock domain)
pi_g_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, gated clock)
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager, free clock)
po_en_clock	output	var reg	Transmit GMII/MII 125/25/2.5 MHz clock gated clock enable
pi_sop	input	wire logic	Signals comming in from the data FIFO related to data transfer Start of data frame indication
pi_eop	input	wire logic	End of data frame indication
pi_pad	input	wire logic	Pad append command, valid only when end of frame (When padding enable
pi_crc	input	wire logic	and frame has less than 64 bytes the CRC is appended regardless of the CRC append setting CRC append command, valid only when end of frame
pi_valid	input	wire logic	Valid data

continues on next page

Table 74 – continued from previous page

Name	Direction	Type	Description
pi_fc_req	input	wire logic	Request & Acknowledge Flow control frame transmit request
pi_data_req	input	wire logic	Data frame transmit request
po_fc_ack	output	var reg	Flow control frame transmit acknowledge
po_data_ack	output	var reg	Data frame transmit acknowledge
po_read	output	var reg	Read & Retransmit & Drop current frame Read next data
po_remove	output	var reg	Remove current frame (drop frame)
po_reload	output	var reg	Reload current frame (retransmit)
po_mac_state	output	var reg[3:0]	Transmit FSM state Transmit FSM state current
po_update	output	var reg	Collision window Collision window (update read pointer)
po_en_high	output	var reg	Enable increment counter
pi_half_duplex	input	wire logic	Half duplex flow control enable Half duplex operating mode
pi_hd_fc_en	input	wire logic	Half-Duplex Flow Control enable
pi_stop_xmit	input	wire logic	Transmit start/stop Transmit EMAC stop command
po_stop_xmit	output	var reg	Transmit EMAC stopped
po_bk_reset	output	var reg	Control Signals Related to the Backoff Block Backoff counter reset command
po_bk_start	output	var reg	Backoff algorithm start command
pi_bk_done	input	wire logic	Backoff done indication
pi_deferring	input	wire logic	Control Signal from the deferral counter Defer current transmition (when asserted high)
pi_deferred	input	wire logic	Deferred frame statistic information (asserted for one cycle)
pi_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_burst_lim	input	wire logic[15:0]	Burst limit (valid only when operating mode is 1000 Mbps)
pi_burst_en	input	wire logic	Burst enable (valid only when operating mode is 1000 Mbps)
po_burst_en	output	var reg	Burst enable (valid only when operating mode is 1000 Mbps)
pi_gmii_en	input	wire logic	GMII Interface Transmit MII/GMII enable indication (to PHY)
pi_gmii_col	input	wire logic	Collision indication (from PHY)
pi_gmii_crs	input	wire logic	Carrier Sense indication (from PHY)
po_xmit_nfc	output	wire logic	(compensate the tx_gmii latency, connected to po_gmii_en on MII/GMII module) Transmit full duplex data frame transmit enable pending (non flow control frame is transmitted) Transmit FSM data frame transmit enable
pi_stat_full	input	wire logic	NOTE: This signal is not asserted during flow control frame transmission Statistic related signals Statistic FIFO full
po_stat_data	output	wire logic[9:0]	Statistic FIFO data
po_stat_load	output	var reg	Statistic FIFO write enable signal

Always Blocks

- always @(posedge pi_g_clock or negedge pi_reset)

MAC State Decoder

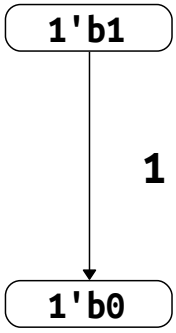


Table 75: FSM Transitions for force_col

#	Current State	Next State	Condition
1	1'b1	1'b0	[(!(~ pi_reset) && (po_mac_state == 4'h1)), (!(~ pi_-reset) && (po_mac_state == 4'h1))]

- always @(posedge pi_f_clock or negedge pi_reset)

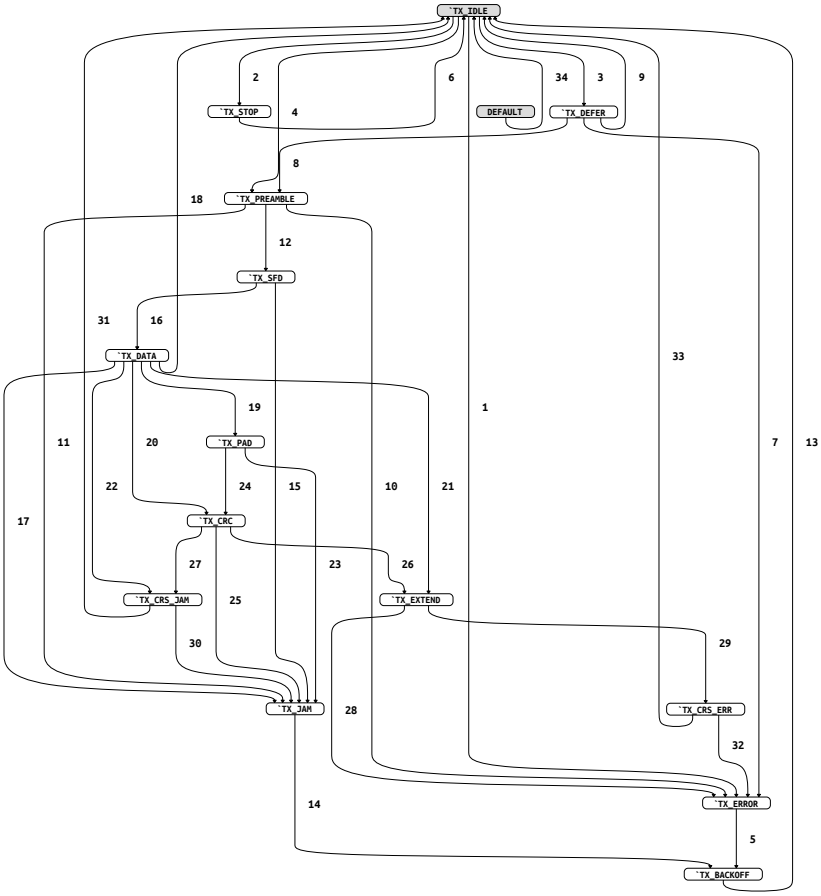


Table 76: FSM Transitions for po_mac_state

#	Current State	Next State	Condition
1	`TX_IDLE	`TX_ERROR	[(!(~ pi_reset) && (gmii_col == 1'b1 && po_burst_en == 1'b1))]
2	`TX_IDLE	`TX_STOP	[(!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && (stop_xmit == 1'b1))]
3	`TX_IDLE	`TX_DEFER	[(!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && !(stop_xmit == 1'b1) && (pi_fc_req == 1'b1)), (!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && !(stop_xmit == 1'b1) && !(pi_fc_req == 1'b1) && (pi_data_req == 1'b1))]
4	`TX_IDLE	`TX_PREAMBLE	[(!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && !(stop_xmit == 1'b1) && !(pi_fc_req == 1'b1) && !(pi_data_req == 1'b1) && (gmii_crs == 1'b1 && gmii_crs_del == 1'b0 && pi_hd_fc_en == 1'b1))]
5	`TX_ERROR	`TX_BACKOFF	[(!(~ pi_reset) && ({counter[1 : 0], po_en_high} == {1'b1, ~ pi_gigabit, pi_gigabit}))]
6	`TX_STOP	`TX_IDLE	[(!(~ pi_reset) && (stop_xmit == 1'b0))]
7	`TX_DEFER	`TX_ERROR	[(!(~ pi_reset) && (gmii_col == 1'b1 && po_burst_en == 1'b1))]
8	`TX_DEFER	`TX_PREAMBLE	[(!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && (pi_deferring == 1'b0 && pi_sop == 1'b1)), (!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && !(pi_deferring == 1'b0 && pi_sop == 1'b1) && !(pi_deferring == 1'b0 && force_col == 1'b1) && (pi_deferring == 1'b0))]
9	`TX_DEFER	`TX_IDLE	[(!(~ pi_reset) && !(gmii_col == 1'b1 && po_burst_en == 1'b1) && !(pi_deferring == 1'b0 && pi_sop == 1'b1) && (pi_deferring == 1'b0 && force_col == 1'b1))]
10	`TX_PREAMBLE	`TX_ERROR	[(!(~ pi_reset) && (gmii_col == 1'b1 && pi_gmii_en == 1'b0))]
11	`TX_PREAMBLE	`TX_JAM	[(!(~ pi_reset) && !(gmii_col == 1'b1 && pi_gmii_en == 1'b0) && (gmii_col == 1'b1))]
12	`TX_PREAMBLE	`TX_SFD	[(!(~ pi_reset) && !(gmii_col == 1'b1 && pi_gmii_en == 1'b0) && !(gmii_col == 1'b1) && ({counter[2 : 1], po_en_high} == {2'd3, 1'b1}))]
13	`TX_BACKOFF	`TX_IDLE	[(!(~ pi_reset) && (col_counter[4] == 1'b1 || window_out == 1'b1)), (!(~ pi_reset) && !(col_counter[4] == 1'b1 || window_out == 1'b1) && (pi_bk_done == 1'b1 && po_bk_start == 1'b0))]
14	`TX_JAM	`TX_BACKOFF	[(!(~ pi_reset) && ({counter[1 : 0], po_en_high} == {1'b1, ~ pi_gigabit | force_col, pi_gigabit | force_col}))]
15	`TX_SFD	`TX_JAM	[(!(~ pi_reset) && (po_en_high == 1'b1 && force_col == 1'b1 || gmii_col == 1'b1))]
16	`TX_SFD	`TX_DATA	[(!(~ pi_reset) && !(po_en_high == 1'b1 && force_col == 1'b1 || gmii_col == 1'b1) && (po_en_high == 1'b1))]
17	`TX_DATA	`TX_JAM	[(!(~ pi_reset) && (gmii_col == 1'b1))]

continues on next page

Table 76 – continued from previous page

#	Current State	Next State	Condition
18	`TX_DATA	`TX_IDLE	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1)), (!(~ pi_reset) && !(gmii_col == 1'b1) && !(counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1) && (pi_valid == 1'b0 && po_en_high == 1'b1))]
19	`TX_DATA	`TX_PAD	[(!(~ pi_reset) && !(gmii_col == 1'b1) && !(counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1) && !(pi_valid == 1'b0 && po_en_high == 1'b1) && (int_eop == 1'b1 && count_60 == 1'b0 && pi_pad == 1'b1 && po_en_high == 1'b1))]
20	`TX_DATA	`TX_CRC	[(!(~ pi_reset) && !(gmii_col == 1'b1) && !(counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1) && !(pi_valid == 1'b0 && po_en_high == 1'b1) && !(int_eop == 1'b1 && count_60 == 1'b0 && pi_pad == 1'b1 && po_en_high == 1'b1) && (int_eop == 1'b1 && pi_crc == 1'b1 && po_en_high == 1'b1))]
21	`TX_DATA	`TX_EXTEND	[(!(~ pi_reset) && !(gmii_col == 1'b1) && !(counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1) && !(pi_valid == 1'b0 && po_en_high == 1'b1) && !(int_eop == 1'b1 && count_60 == 1'b0 && pi_pad == 1'b1 && po_en_high == 1'b1) && !(int_eop == 1'b1 && pi_crc == 1'b1 && po_en_high == 1'b1) && (int_eop == 1'b1 && count_512 == 1'b0 && burst_begin == 1'b1 && pi_half_duplex == 1'b1))]
22	`TX_DATA	`TX_CRS_JAM	[(!(~ pi_reset) && !(gmii_col == 1'b1) && !(counter[14] == 1'b1 && counter[0] == 1'b1 && po_en_high == 1'b1) && !(pi_valid == 1'b0 && po_en_high == 1'b1) && !(int_eop == 1'b1 && count_60 == 1'b0 && pi_pad == 1'b1 && po_en_high == 1'b1) && !(int_eop == 1'b1 && pi_crc == 1'b1 && po_en_high == 1'b1) && !(int_eop == 1'b1 && count_512 == 1'b0 && burst_begin == 1'b1 && pi_half_duplex == 1'b1) && (int_eop == 1'b1 && po_en_high == 1'b1))]
23	`TX_PAD	`TX_JAM	[(!(~ pi_reset) && (gmii_col == 1'b1))]
24	`TX_PAD	`TX_CRC	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (count_60 == 1'b1 && po_en_high == 1'b1))]
25	`TX_CRC	`TX_JAM	[(!(~ pi_reset) && (gmii_col == 1'b1))]
26	`TX_CRC	`TX_EXTEND	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (crc_-counter == 2'd3 && count_512 == 1'b0 && burst_-begin == 1'b1 && pi_half_duplex == 1'b1))]
27	`TX_CRC	`TX_CRS_JAM	[(!(~ pi_reset) && !(gmii_col == 1'b1) && !(crc_-counter == 2'd3 && count_512 == 1'b0 && burst_be-gin == 1'b1 && pi_half_duplex == 1'b1) && ({crc_-counter, po_en_high} == {2'd3, 1'b1}))]
28	`TX_EXTEND	`TX_ERROR	[(!(~ pi_reset) && (gmii_col == 1'b1))]
29	`TX_EXTEND	`TX_CRS_ERR	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (count_512 == 1'b1))]
30	`TX_CRS_JAM	`TX_JAM	[(!(~ pi_reset) && (gmii_col == 1'b1))]
31	`TX_CRS_JAM	`TX_IDLE	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (gmii_crs == 1'b0 || po_burst_en == 1'b1))]

continues on next page

Table 76 – continued from previous page

#	Current State	Next State	Condition
32	`TX_CRS_ERR	`TX_ERROR	[(!(~ pi_reset) && (gmii_col == 1'b1))]
33	`TX_CRS_ERR	`TX_IDLE	[(!(~ pi_reset) && !(gmii_col == 1'b1) && (gmii_crs == 1'b0 || po_burst_en == 1'b1))]
34	default	`TX_IDLE	[!(~ pi_reset)]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_fsm`

6.39 Module ip_mac_tx_gmii_g

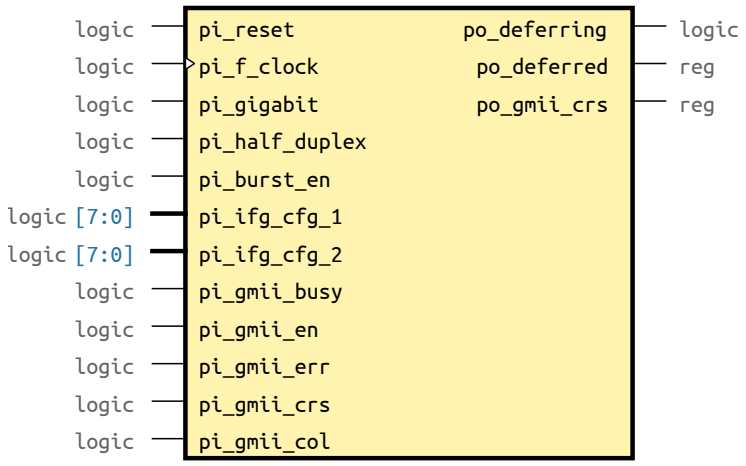


Fig. 51: Block Diagram of ip_mac_tx_gmii_g

Overview

Frames are transmitted over the EMAC MII/GMII interface with an interframe gap which is specified by the IEEE 802.3-2002 standard to be 96 bit times (9.6 us for 10 Mbps, 0.96 us for 100 Mbps, 0.096 us for 1000 Mbps). This is a minimum value and may be increased with a resulting decrease in throughput (results in a less aggressive approach to gain access to a shared Ethernet bus). The process for deferring the transmission is different for half-duplex and full-duplex systems and is as follows:

Half-Duplex Even when it has nothing to transmit, the EMAC monitors the bus for traffic by watching the carrier sense signal from the external PHY. Whenever the bus is busy, the EMAC defers to the passing frame by delaying any pending transmission of its own. After the last bit of the passing frame (when carrier sense signal changes from true to false), the EMAC starts the timing of the interframe gap. The EMAC will reset the interframe gap timer if carrier sense becomes true during the first period defined by the IFG1. The IEEE 802.3-2002 standard states that this should be the first 2/3 of the interframe gap interval (64 bit times) but may be shorter and as small as zero. The EMAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by the second IFG2 field to ensure fair access to the bus. The IEEE 802.3-2002 standard states that this should be the last 1/3 of the interframe gap timing interval (32 bit times) but may be longer and as large as the whole interframe gap time.

Full-Duplex The EMAC does not use the carrier sense signal from the external PHY when in full duplex mode since the bus is not shared and only needs to monitor its own transmissions. After the last bit of an EMAC transmission, the EMAC starts the interframe gap timer and defers transmissions until it has reached the value represented by the combination of the IFG1 and IFG2.

The EMAC Transmit MII/GMII module is responsible also to demultilex the GMII interface signals coming from EMAC Transmit State Machine into MII format (nibble oriented) when 10/100 Mbps operating mode is selected or to pass the GMII signal to the output when 1000 Mbps operating mode. The GMII/MII block is responsible to register the input signals comming from the physical layer, and synchronize the carrier sense indication (two DFF's are required since the carrier sense is an asynchronous input)

Table 77: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Global Hardware/Software reset (active low)

continues on next page

Table 77 – continued from previous page

Name	Direction	Type	Description
pi_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_half_duplex	input	wire logic	Defer interface Operating Half Duplex mode
pi_burst_en	input	wire logic	Burst enable (valid only when operating mode is 1000 Mbps)
pi_ifg_cfg_1	input	wire logic[7:0]	Interframe gap part 1 (usualy 2/3 from IFG)
pi_ifg_cfg_2	input	wire logic[7:0]	Interframe gap part 2 (usualy 1/3 from IFG)
po_deferring	output	wire logic	Defer current transmition (when asserted high)
po_deferred	output	var reg	Statistic information (frame was deferred)
pi_gmii_busy	input	wire logic	Tx MAC state interface
pi_gmii_en	input	wire logic	Transmit GMII enable indication
pi_gmii_err	input	wire logic	Transmit GMII error indication
po_gmii_crs	output	var reg	Carrier Sense indication
pi_gmii_crs	input	wire logic	MII Transmit Enable and Data Output Signals, Carrier Sense and Colision Indicators Carrier Sense indication Asynchronous input (2 DFF required for synchronization) (from PHY)
pi_gmii_col	input	wire logic	Collision indication (from PHY)

Always Blocks

- always @(posedge pi_f_clock or negedge pi_reset)

Defer Counter

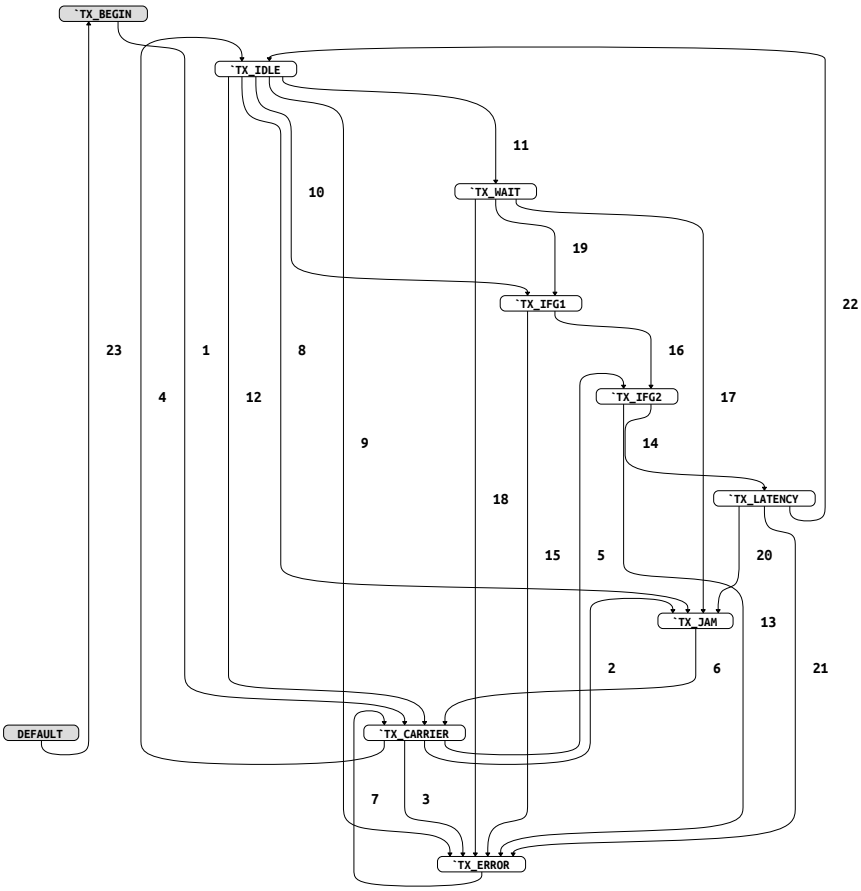


Table 78: FSM Transitions for defer_state

#	Current State	Next State	Condition
1	`TX_BEGIN	`TX_CARRIER	[EMPTY]
2	`TX_CARRIER	`TX_JAM	[(pi_gmii_en == 1'b1 && pi_gmii_busy == 1'b1 && gmii_col == 1'b1)]
3	`TX_CARRIER	`TX_ERROR	[(!(pi_gmii_en == 1'b1 && pi_gmii_busy == 1'b1 && gmii_col == 1'b1) && (pi_gmii_err == 1'b1 && pi_-gmii_busy == 1'b1 && gmii_col == 1'b1))]
4	`TX_CARRIER	`TX_IDLE	[(!(pi_gmii_en == 1'b1 && pi_gmii_busy == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && pi_-gmii_busy == 1'b1 && gmii_col == 1'b1) && (pi_-gmii_busy == 1'b1))]
5	`TX_CARRIER	`TX_IFG2	[(!(pi_gmii_en == 1'b1 && pi_gmii_busy == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && pi_gmii_busy == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_busy == 1'b1) && !(gmii_crs == 1'b1) && (counter == 9'd0))]
6	`TX_JAM	`TX_CARRIER	[(counter == 9'd1)]
7	`TX_ERROR	`TX_CARRIER	[(counter == 9'd1)]
8	`TX_IDLE	`TX_JAM	[(pi_gmii_en == 1'b1 && gmii_col == 1'b1)]
9	`TX_IDLE	`TX_ERROR	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && (pi_gmii_err == 1'b1 && gmii_col == 1'b1))]

continues on next page

Table 78 – continued from previous page

#	Current State	Next State	Condition
10	`TX_IDLE	`TX_IFG1	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_busy == 1'b1) && (pi_gmii_busy == 1'b0 && gmii_busy_del == 1'b1 && gmii_crs == 1'b0 || pi_burst_en == 1'b1))]
11	`TX_IDLE	`TX_WAIT	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_busy == 1'b1) && !(pi_gmii_busy == 1'b0 && gmii_busy_del == 1'b1 && gmii_crs == 1'b0 || pi_burst_en == 1'b1) && (pi_gmii_busy == 1'b0 && gmii_busy_del == 1'b1))]
12	`TX_IDLE	`TX_CARRIER	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_busy == 1'b1) && !(pi_gmii_busy == 1'b0 && gmii_busy_del == 1'b1 && gmii_crs == 1'b0 || pi_burst_en == 1'b1) && !(pi_gmii_busy == 1'b0 && gmii_busy_del == 1'b1) && (gmii_crs == 1'b1 && pi_gmii_en == 1'b0 && pi_gmii_err == 1'b0))]
13	`TX_IFG2	`TX_ERROR	[(pi_gmii_err == 1'b1 && gmii_col == 1'b1)]
14	`TX_IFG2	`TX_LATENCY	[(!(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && (counter == 9'd0))]
15	`TX_IFG1	`TX_ERROR	[(pi_gmii_err == 1'b1 && gmii_col == 1'b1)]
16	`TX_IFG1	`TX_IFG2	[(!(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && (counter == 9'd0))]
17	`TX_WAIT	`TX_JAM	[(pi_gmii_en == 1'b1 && gmii_col == 1'b1)]
18	`TX_WAIT	`TX_ERROR	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && (pi_gmii_err == 1'b1 && gmii_col == 1'b1))]
19	`TX_WAIT	`TX_IFG1	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && (gmii_crs == 1'b0))]
20	`TX_LATENCY	`TX_JAM	[(pi_gmii_en == 1'b1 && gmii_col == 1'b1)]
21	`TX_LATENCY	`TX_ERROR	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && (pi_gmii_err == 1'b1 && gmii_col == 1'b1))]
22	`TX_LATENCY	`TX_IDLE	[(!(pi_gmii_en == 1'b1 && gmii_col == 1'b1) && !(pi_gmii_err == 1'b1 && gmii_col == 1'b1) && (counter == 9'd0))]
23	default	`TX_BEGIN	[EMPTY]

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_gmii`

6.40 Module ip_mac_tx_sync_g

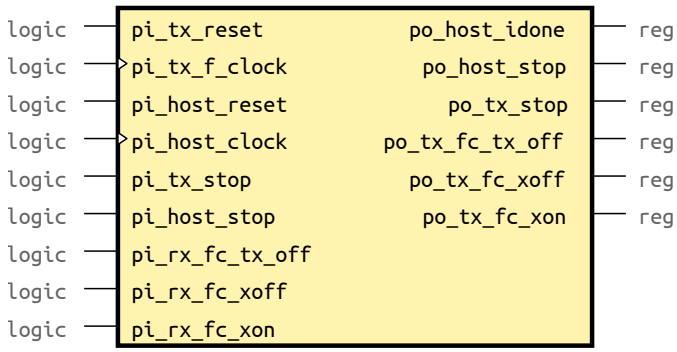


Fig. 52: Block Diagram of ip_mac_tx_sync_g

Overview

The EMAC Transmit Synchronization module is responsible to synchronize all the control signals necessary to control the functionality of the transmit clock domain modules. Also the signals generated by the transmit modules and needed by the host clock domain modules are synchronized by the EMAC Transmit Synchronization module to host clock domain. In order to avoid the metastability of the output signal the synchronization from one clock domain to another will require two DFF's. The host and transmit clocks used by the synchronization module are free running clocks, since there is no possibility to generate synchronous wake-up's signals for gated clock module. Also this module is responsible to synchronize the reset done information (initialization done) used by the host DMA module in order to start data transfers.

Table 79: Ports

Name	Direction	Type	Description
pi_tx_reset	input	wire logic	Transmit clock and reset Global Hardware/-Software reset (transmit clock domain, active low)
pi_tx_f_clock	input	wire logic	Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_host_idone	output	var reg	Reset done Transmit initialization done (host clock domain)
pi_host_reset	input	wire logic	Host clock and reset Global Hardware/Software reset (host clock domain, active low)
pi_host_clock	input	wire logic	Host clock (from Clock Manager)
pi_tx_stop	input	wire logic	Inputs -> Synchronized outputs Transmit stop command acknowledge (transmit clock domain)
po_host_stop	output	var reg	Transmit stop command acknowledge (synchronized to host clock domain)
pi_host_stop	input	wire logic	Transmit stop command (host clock domain)
po_tx_stop	output	var reg	Transmit stop command (synchronized to transmit clock domain)
pi_rx_fc_tx_off	input	wire logic	Received transmit off command (receive clock domain)
po_tx_fc_tx_off	output	var reg	Transmit off command (synchronized to transmit clock domain)

continues on next page

Table 79 – continued from previous page

Name	Direction	Type	Description
pi_rx_fc_xoff	input	wire logic	Receive FIFO level exceed high threshold, XOFF FC packet insert (receive clock domain)
pi_rx_fc_xon	input	wire logic	Transmit XOFF FC packet insert (synchronized to transmit clock domain)
po_tx_fc_xoff	output	var reg	Receive FIFO level below low threshold, XON FC packet insert (receive clock domain)
po_tx_fc_xon	output	var reg	Transmit XON FC packet insert (synchronized to transmit clock domain)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > `tx_sync`

6.41 Module ip_mac_tx_top_g

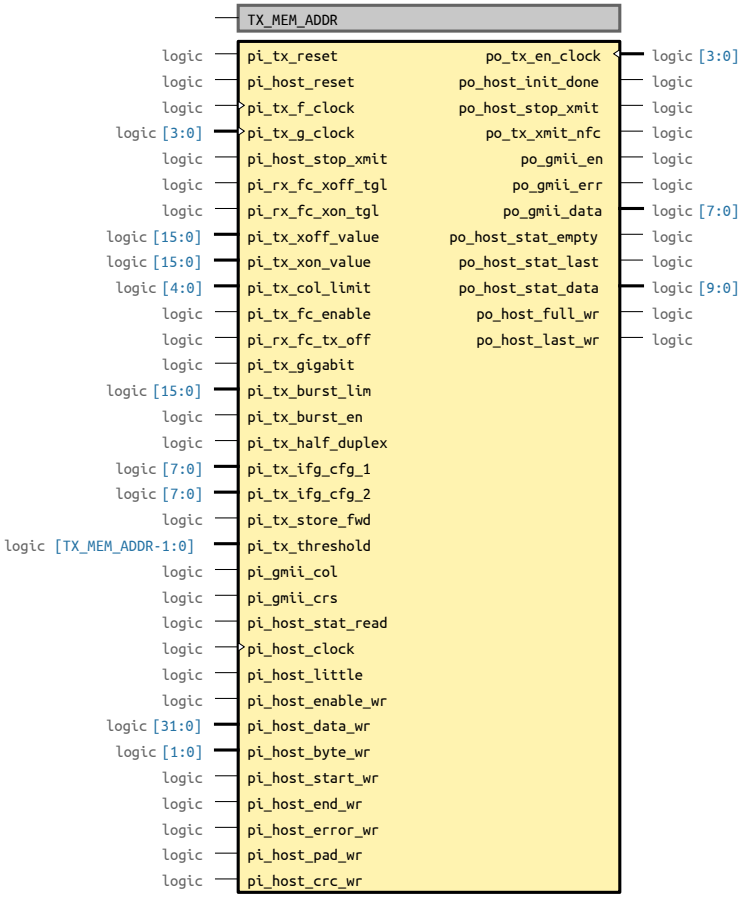


Fig. 53: Block Diagram of ip_mac_tx_top_g

Table 80: Parameters

Name	Default value	Description
TX_MEM_ADDR	9	Transmit memory address width (9 -> 512 locations, 10->1024)

Table 81: Ports

Name	Direction	Type	Description
pi_tx_reset	input	wire logic	Global Software/Hardware Reset (transmit clock domain)
pi_host_reset	input	wire logic	Global Software/Hardware Reset (host clock domain)
pi_tx_f_clock	input	wire logic	Transmit clock Free Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
pi_tx_g_clock	input	wire logic[3:0]	Gated Transmit GMII/MII 125/25/2.5 MHz clock (from Clock Manager)
po_tx_en_clock	output	wire logic[3:0]	Enable Transmit GMII/MII 125/25/2.5 MHz clock gated clock enable
po_host_init_done	output	wire logic	Initialisation done Transmit initialisation done (host clock domain)

continues on next page

Table 81 – continued from previous page

Name	Direction	Type	Description
pi_host_stop_xmit	input	wire logic	Start/Stop transmit process Transmit EMAC stop command
po_host_stop_xmit	output	wire logic	Transmit EMAC stopped
pi_rx_fc_xoff_tgl	input	wire logic	Configuration Interface from RX EMAC insert XOFF flow control information
pi_rx_fc_xon_tgl	input	wire logic	from RX EMAC insert XON flow control information
pi_tx_xoff_value	input	wire logic[15:0]	from configuration XOFF flow control pause value
pi_tx_xon_value	input	wire logic[15:0]	from configuration XON flow control pause value
pi_tx_col_limit	input	wire logic[4:0]	Half Duplex back presure collision limit
pi_tx_fc_enable	input	wire logic	(maximum collision number during back presure algorithm) Transmit flow control enable
pi_rx_fc_tx_off	input	wire logic	Flow control Stop transmit commang (flow control received)
pi_tx_gigabit	input	wire logic	Operating 1000 Mbps (Gigabit) mode
pi_tx_burst_lim	input	wire logic[15:0]	Burst limit (valid only when operating mode is 1000 Mbps)
pi_tx_burst_en	input	wire logic	Burst enable (valid only when operating mode is 1000 Mbps)
pi_tx_half_duplex	input	wire logic	Operating Half Duplex mode
pi_tx_ifg_cfg_1	input	wire logic[7:0]	Interframe gap part 1 (usualy 2/3 from IFG)
pi_tx_ifg_cfg_2	input	wire logic[7:0]	Interframe gap part 2 (usualy 1/3 from IFG)
pi_tx_store_fwd	input	wire logic	Store and Forward transmit FIFO operating mode
pi_tx_threshold	input	wire logic[TX_-MEM_ADDR-1:0]	Cut Trough (pi_emac_store_fwd not asserted) FIFO threshold
po_tx_xmit_nfc	output	wire logic	GMII/MII interface Transmit FSM data frame transmit enable (transmit clock domain)
po_gmii_en	output	wire logic	NOTE: This signal is not asserted during flow control frame transmission Transmit MII/GMII enable indication (to PHY)
po_gmii_err	output	wire logic	Transmit MII/GMII error indication (to PHY)
po_gmii_data	output	wire logic[7:0]	Transmit MII/GMII data (MII data is po_-emac_tx_data[3:0]) (to PHY)
pi_gmii_col	input	wire logic	Collision indication (from PHY)
pi_gmii_crs	input	wire logic	Carrier Sense indication (from PHY)
pi_host_stat_read	input	wire logic	Read statistic word command
po_host_stat_-empty	output	wire logic	Statistic FIFO not empty
po_host_stat_last	output	wire logic	Last statistic word indication
po_host_stat_data	output	wire logic[9:0]	Statistic TDES0 word data
pi_host_clock	input	wire logic	HOST interface HOST interface clock signal
pi_host_little	input	wire logic	Little endian (data path organisation)
pi_host_enable_wr	input	wire logic	Transmit MAC data path, HOST enable command
po_host_full_wr	output	wire logic	Transmit MAC data path, HOST FIFO full indication

continues on next page

Table 81 – continued from previous page

Name	Direction	Type	Description
po_host_last_wr	output	wire logic	Transmit MAC data path, HOST FIFO last location indication
pi_host_data_wr	input	wire logic[31:0]	Transmit MAC data path, HOST data (transmit data)
pi_host_byte_wr	input	wire logic[1:0]	Transmit MAC data path, HOST byte enable (transmit data byte enable)
pi_host_start_wr	input	wire logic	Transmit MAC data path, HOST start of frame indication
pi_host_end_wr	input	wire logic	Transmit MAC data path, HOST start of frame indication
pi_host_error_wr	input	wire logic	Unused please check if useful (if not remove)
pi_host_pad_wr	input	wire logic	Transmit MAC data path, HOST padding enable (valid only when pi_host_end_wr)
pi_host_crc_wr	input	wire logic	Transmit MAC data path, HOST crc enable (valid only when pi_host_end_wr)

Instances

- *ip_emac_top* > *ip_mac_top_g* > `mac_tx_top : ip_mac_tx_top_g#(.TX_MEM_ADDR(10))`

Submodules

- **ip_mac_tx_top_g#(.TX_MEM_ADDR(10))**
  - tx_bkoff : *ip_mac_tx_bkoff_g*
  - tx_data_async : *ip_async_fifo_g* #(.MEM_WIDTH(39))
  - tx_data_dram : *ip_mac_dram_001* #(.MEM_ADDR(10), .MEM_WIDTH(39))
  - tx_dpath : *ip_mac_tx_dpath_g*
  - tx_dsplit : *ip_mac_tx_dsplit_g* #(.MEM_ADDR(10))
  - tx_endian : *ip_mac_big_endian*
  - tx_fc_gen : *ip_mac_fc_gen_g*
  - tx_fifo : *ip_mac_tx_fifo_g* #(.MEM_ADDR(10))
  - tx_fsm : *ip_mac_tx_fsm_g*
  - tx_gmii : *ip_mac_tx_gmii_g*
  - tx_stat_async : *ip_async_fifo_g* #(.MEM_WIDTH(10))
  - tx_sync : *ip_mac_tx_sync_g*

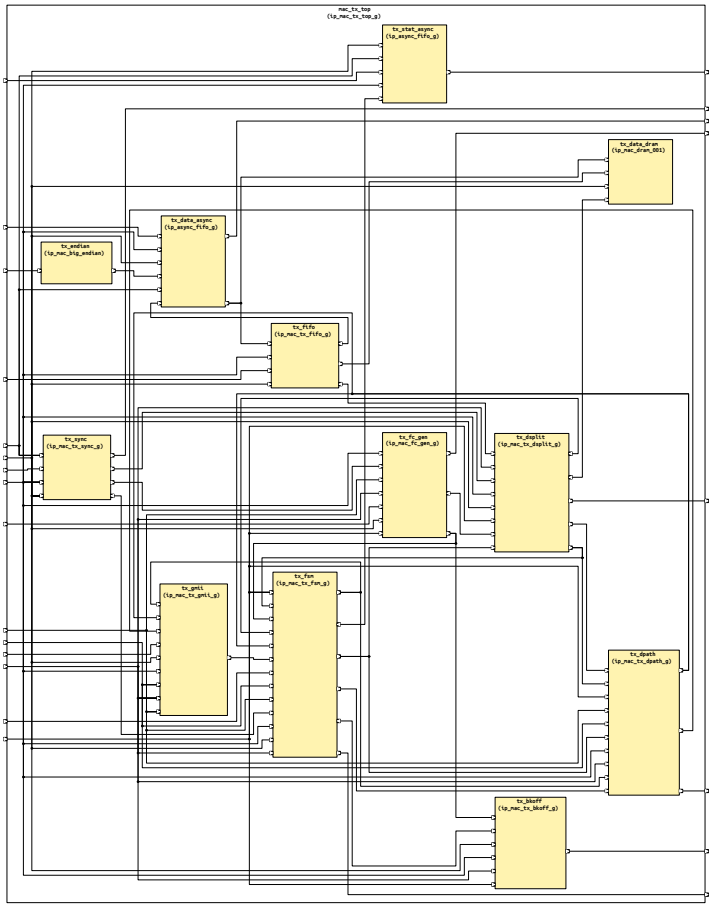


Fig. 54: Flow Diagram of ip_mac_tx_top_g

6.42 Module ip_sync_cell

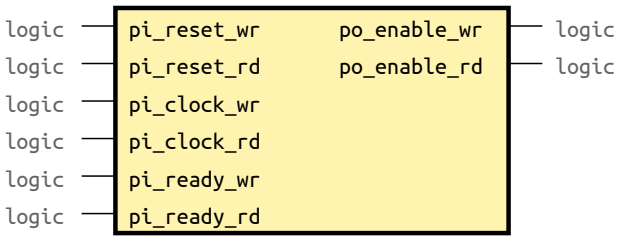


Fig. 55: Block Diagram of ip_sync_cell

Overview

The synchronization cell is responsible for the write/read enable signals synchronization. The write enable signal indicates that a new write clock domain data can be synchronized to the read clock domain. The read enable signal indicates that the external data can be safely read on the read clock domain.

Table 82: Ports

Name	Direction	Type	Description
pi_reset_wr	input	wire logic	reset write clock domain (synchronous)
pi_reset_rd	input	wire logic	reset read clock domain (synchronous)
pi_clock_wr	input	wire logic	write clock
pi_clock_rd	input	wire logic	read clock
po_enable_wr	output	wire logic	write enable (combinatorial output)
po_enable_rd	output	wire logic	read enable (combinatorial output)
pi_ready_wr	input	wire logic	write ready (data available for write)
pi_ready_rd	input	wire logic	read ready (data available for read)

Instances

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_2`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_3`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_4`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_5`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_6`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_rx_top_g* > *ip_async_fifo_g* > `cell_7`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_2`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_3`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_4`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_5`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_6`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_7`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_0`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_1`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_2`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_3`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_4`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_5`

- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_6`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_tx_top_g* > *ip_async_fifo_g* > `cell_7`

Submodules

- **ip_sync_cell**
  - metastable_toggle_rd : *dff_metastable* #(.DFF_WIDTH(1))
  - metastable_toggle_wr : *dff_metastable* #(.DFF_WIDTH(1))

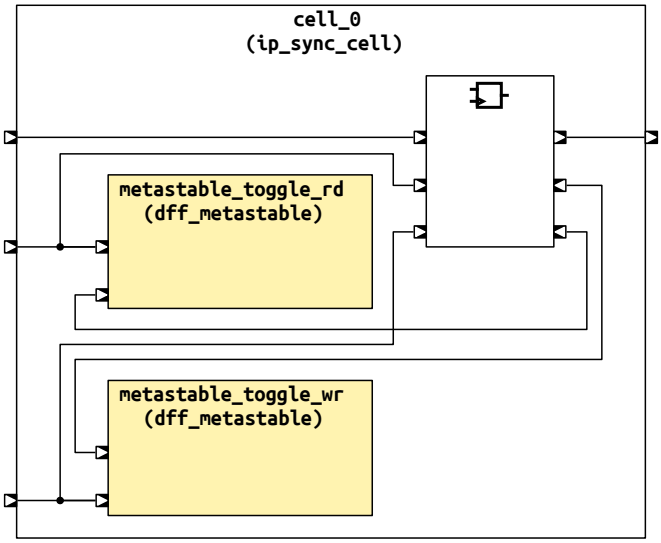


Fig. 56: Flow Diagram of ip_sync_cell

6.43 Module ip_sync_reset_g

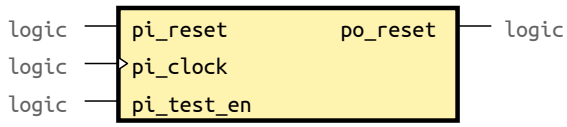


Fig. 57: Block Diagram of ip_sync_reset_g

Overview

Synchronous resets are used in designs to safely reset the DFFs from one clock domain. The asynchronous reset can generate an unstable condition in design due to the metastability when reset is removed, if the DFF input data is different than the reset value output DFF data (for example a free counter)

Table 83: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	Asynchronous reset
pi_clock	input	wire logic	Input clock
pi_test_en	input	wire logic	Test enable (multiplex information)
po_reset	output	wire logic	Synchronous reset (functional clock balanced)

Instances

- *ip_emac_top* > *ip_host_clk_mng_g* > `host_sreset`
- *ip_emac_top* > *ip_host_clk_mng_g* > `hw_host_sreset`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `host_sreset`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `hw_host_sreset`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `mdio_sreset`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `rx_sreset`
- *ip_emac_top* > *ip_mac_top_g* > *ip_mac_clk_mng_g* > `tx_sreset`

6.44 Module ip_synchronous_fifo

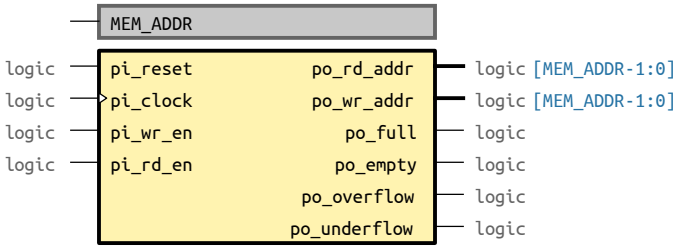


Fig. 58: Block Diagram of ip_synchronous_fifo

Table 84: Parameters

Name	Default value	Description
MEM_ADDR	6	Transmit memory address width (9 -> 512 locations, 10->1024)

Table 85: Ports

Name	Direction	Type	Description
pi_reset	input	wire logic	
pi_clock	input	wire logic	
po_rd_addr	output	wire logic[MEM_-ADDR-1:0]	
po_wr_addr	output	wire logic[MEM_-ADDR-1:0]	
pi_wr_en	input	wire logic	
pi_rd_en	input	wire logic	
po_full	output	wire logic	
po_empty	output	wire logic	
po_overflow	output	wire logic	
po_underflow	output	wire logic	

CHAPTER
SEVEN

MACROS

Table 1: Macros

Name	Type	Value	Description
AE_BIT	defined	2	
ARB_IDLE	defined	3'b000	ARBITER states
ARB_RX	defined	3'b010	
ARB_RX_DS	defined	3'b100	
ARB_S0	defined	3'b001	
ARB_TX	defined	3'b011	
ARB_TX_DS	defined	3'b101	
ARB_TX_UPD	defined	3'b110	
BF_BIT	defined	9	
BK_DONE	defined	1'b1	Backoff DONE state (wait slot_cnt tobe equal bk_ended value)
BK_IDLE	defined	1'b0	Backoff IDLE state (wait for pi_bk_start command)
CC_BIT	defined	5:2	
CDID_DEVICE_-TYPE	defined	16'h3030	
CE_BIT	defined	1	
CFID_MAC_ID	defined	16'h1010	
CFID_MANU-FACTURER_ID	defined	16'h2020	
DATA	defined	3'h7	
DATA_DROP	defined	3'h2	
DATA_READ	defined	3'h1	
DEVADDR	defined	3'h4	
DE_BIT	defined	0	
EC_BIT	defined	6	
EF_BIT	defined	4	
EMAC_10_100_-MBPS	ifdef		
ERROR_-LENGTH	defined	8'h04	
FC_DEST_ADDR	defined	48'h0180C2000001	Control frame destination address
FC_IDLE	defined	2'd0	FSM states encoding (flow control frame FSM)
FC_OP_TYPE	defined	2'd1	
FC_READ	defined	3'h4	
FC_TIME_VAL	defined	2'd2	
FC_WAIT_64	defined	2'd3	
FL_BIT	defined	22:8	Statistic word bits
HASH	defined	3'd2	
HOST_NOP	defined	2'b00	
HOST_READ	defined	2'b10	HOST bus interface operations
HOST_WRITE	defined	2'b01	

continues on next page

Table 1 – continued from previous page

Name	Type	Value	Description
IDLE	defined	3'd0	FSM States And Control Data Define *
LC_BIT	defined	7	
LE_BIT	defined	3	
LP_BIT	defined	7	
MATCH_1	defined	3'd3	
MATCH_2	defined	3'd4	
MF_BIT	defined	6	
MULTICAST_BIT	defined	40	Global multicast bit (see 802.3-2002_part2.pdf, 22.2.3 Frame structure)
OF_BIT	defined	0	
OPCODE	defined	3'h3	
PREAMBLE	defined	3'h1	
REGADDR	defined	3'h5	
REGS_HWRITE	defined	3'b100	
REGS_IDLE	defined	3'b000	
REGS_READ	defined	3'b010	
REGS_WRITE	defined	3'b001	
RX_CRC_CHECK	defined	32'hC704DD7B	CRC check value
RX_DA	defined	4'h3	
RX_DATA_0	defined	4'h6	
RX_DATA_1	defined	4'h7	
RX_DS_IDLE	defined	3'b000	RX ds aquire state values
RX_DS_MAIN	defined	3'b001	
RX_DS_READ	defined	3'b010	
RX_DS_SUS-PEND	defined	3'b011	
RX_DS_WAIT	defined	3'b100	
RX_ERROR_-DATA	defined	8'h1f	
RX_EXTEND	defined	4'h8	
RX_EXTEND_-DATA	defined	8'h0f	
RX_FCOPTYPE_-CHECK_0	defined	8'h88	FC frame opcode
RX_FCOPTYPE_-CHECK_1	defined	8'h08	
RX_FCOPTYPE_-CHECK_2	defined	8'h00	
RX_FCOPTYPE_-CHECK_3	defined	8'h01	
RX_IDLE	defined	4'h0	FSM states encoding (receive frame)
RX_PREAMBLE	defined	4'h1	
RX_PREAM-BLE_DATA	defined	4'h5	GMII Data encoding
RX_READ_DS	defined	3'b001	
RX_SA	defined	4'h4	
RX_SFD	defined	4'h2	
RX_SFD_DATA	defined	4'hd	
RX_STAT	defined	3'b101	
RX_TYPE	defined	4'h5	
RX_WAIT	defined	4'h9	
RX_WRITE	defined	3'b011	
RX_WRITE_DS	defined	3'b110	

continues on next page

Table 1 – continued from previous page

Name	Type	Value	Description
RX_WRITE_-PAUSE	defined	3'b010	
SIMULATION_-VALUE_RE-DUCED	ifdef		synopsys translate_off Reduced 512 time slot (for simulation)
SPLIT_IDLE	defined	3'h0	FSM states encoding
SPLIT_WAIT	defined	3'h3	
START	defined	3'h2	
TJ_BIT	defined	8	
TL_BIT	defined	5	
TP	defined	1	Delay assertion output signals
TURNAR	defined	3'h6	
TX_BACKOFF	defined	4'h8	
TX_BEGIN	defined	4'd0	Defer State Coding * Transmit IFG initialize after reset (only)
TX_CARRIER	defined	4'd7	Carrier monitor (after remote transmit)
TX_CRC	defined	4'h6	
TX_CRS_ERR	defined	4'hd	
TX_CRS_JAM	defined	4'hc	
TX_DATA	defined	4'h4	
TX_DEFER	defined	4'h1	
TX_DS_IDLE	defined	3'b000	TX ds aquire state values
TX_DS_MAIN	defined	3'b001	
TX_DS_READ	defined	3'b010	
TX_DS_SUS-PEND	defined	3'b011	
TX_DS_UPD_-IDLE	defined	2'b00	
TX_DS_UPD_-WRITE	defined	2'b01	
TX_DS_WAIT	defined	3'b100	
TX_ERROR	defined	4'd3	Collision during carrier extend phase
TX_ERROR_-DATA	defined	8'h1f	
TX_EXTEND	defined	4'h9	
TX_EXTEND_-DATA	defined	4'hf	
TX_IDLE	defined	4'd1	Transmit enable
TX_IDLE_DATA	defined	8'h00	
TX_IFG1	defined	4'd5	IFG1 (ignore the carrier sense assertion)
TX_IFG2	defined	4'd6	IFG2 (ignore the carrier sense assertion)
TX_JAM	defined	4'd2	Collision during data phase
TX_JAM_DATA	defined	4'hf	
TX_JAM_NIB-BLE	defined	4'hf	
TX_LATENCY	defined	4'd8	MAC latency 2 cycles
TX_LW_BUFF	defined	4'b1000	
TX_LW_BURST	defined	4'b1001	
TX_LW_DS	defined	4'b0111	
TX_LW_FRM	defined	4'b0110	
TX_MAC_JAB-BER_WORDS_-LIMIT	defined	16'h1001	
TX_PAD	defined	4'h5	

continues on next page

Table 1 – continued from previous page

Name	Type	Value	Description
TX_PAD_DATA	defined	4'h0	
TX_PREAMBLE	defined	4'h2	
TX_PREAM-BLE_DATA	defined	4'h5	
TX_READ	defined	4'b0011	
TX_READ_DS	defined	4'b0001	
TX_READ_-PAUSE	defined	4'b0010	
TX_SFD	defined	4'h3	
TX_SFD_DATA	defined	4'hd	
TX_STOP	defined	4'hb	
TX_UPD_FIFO_-RANGE	defined	3	range of the tx descriptor update fifo =log2(TX_UPD_FIFO_SIZE)
TX_UPD_FIFO_-SIZE	defined	8	size of the tx descriptor update fifo
TX_UPD_WAIT	defined	4'b0101	
TX_WAIT	defined	4'd4	Wait carrier to be deasserted after own transmit
TX_WRITE_DS	defined	4'b0100	
UF_BIT	defined	1	
WAIT	defined	3'd1	
WRITE_1	defined	3'd1	Write table (write odd word)
WRITE_2	defined	3'd2	and exact address match 16-bits (when last Hash + 1 Exact Address Match) Write table exact address match 16-bits (1 Exact Address Match)
WRITE_3	defined	3'd3	Write table exact address match 16-bits (1 Exact Address Match)
WRITE_4	defined	3'd4	Write table exact address match 16-bits (1 Exact Address Match)
WR_DATA	defined	3'd1	
WR_EXTEND	defined	3'd4	
WR_IDLE	defined	3'd0	FSM states encoding (memory write FSM)
WR_OVERRUN	defined	3'd2	
WR_STAT	defined	3'd3	